Bootstraps, Permutation Tests, and Sampling Orders of Magnitude Faster Using SAS®

John Douglas ("J.D.") Opdyke*

Abstract

While permutation tests and bootstraps have very wide-ranging application, both share a common potential drawback: as data-intensive resampling methods, both can be runtime prohibitive when applied to large or even medium-sized data samples drawn from large datasets. The data explosion over the past few decades has made this a common occurrence, and it highlights the increasing need for faster, and more efficient and scalable, permutation test and bootstrap algorithms.

Seven bootstrap and six permutation test algorithms coded in SAS (the largest privately owned software firm globally) are compared. The fastest algorithms ("OPDY" for the bootstrap, "OPDN" for permutation tests) are new, use no modules beyond Base SAS, and achieve speed increases orders of magnitude faster than the relevant "built-in" SAS procedures (OPDY is over 200x faster than Proc SurveySelect; OPDN is over 240x faster than Proc SurveySelect, over 350x faster than NPAR1WAY (which crashes on datasets less than a tenth the size OPDN can handle), and over 720x faster than Proc Multtest). OPDY also is much faster than hashing, which crashes on datasets smaller – sometimes by orders of magnitude – than OPDY can handle. OPDY is easily generalizable to multivariate regression models, and OPDN, which uses an extremely efficient draw-by-draw random-sampling-without-replacement algorithm, can use virtually any permutation statistic, so both have a very wide range of application. And the time complexity of both OPDY and OPDN is sub-linear, making them not only the fastest, but also the only truly scalable bootstrap and permutation test algorithms, respectively, in SAS.

Keywords:
Bootstrap, Permutation, SAS, Scalable, Hashing, With Replacement, Without Replacement, Sampling

JEL Classifications: C12, C13, C14, C15, C63, C88
Mathematics Subject Classifications: 62F40, 62G09, 62G10

* J.D. Opdyke is Managing Director, DataMineit, LLC, a consultancy specializing in advanced statistical and econometric modeling, risk analytics, and algorithm development for the banking, finance, and consulting sectors. J.D. has been a SAS user for over 20 years and routinely writes SAS code faster (often orders of magnitude faster) than SAS Procs (including but not limited to Proc Logistic, Proc MultTest, Proc Summary, Proc NPAR1WAY, Proc Freq, Proc Plan, and Proc SurveySelect). He earned his undergraduate degree with honors from Yale University, his graduate degree from Harvard University where he was both a Kennedy Fellow and a Social Policy Research Fellow, and he has completed post-graduate work as an ASP Fellow in the graduate mathematics department at MIT. His peer reviewed publications span number theory/combinatorics, robust statistics and high-convexity VaR modeling for regulatory and economic capital estimation, statistical finance, statistical computation, applied econometrics, and hypothesis testing for statistical quality control. Most are available upon request from J.D. at jdopdyke@datamineit.com.

Introduction

Permutation tests are as old as modern statistics (see Fisher, 1935), and their statistical properties are well understood and well documented in the literature (see Mielke & Berry, 2001, and Pesarin, 2001, for comprehensive treatments and extensive bibliographies[1]). They typically are nonparametric hypothesis tests

[1] Other widely used texts include Edgington (2007), Good (2004), and Manly (2007).

that avoid making restrictive, and often unrealistic parametric (distributional) assumptions about the data being tested.[2] As such, permutation tests are the most appropriate statistic of choice in a very wide range of scientific and industry settings.

Since its more recent introduction just over three decades ago, the bootstrap (see Efron, 1979) has become a ubiquitous statistical procedure used mainly to obtain standard errors or confidence intervals for statistical estimates. Its popularity, too, is largely due to the fact that it typically is applied nonparametrically, allowing analyses unencumbered by restrictive and often unrealistic distributional data assumptions or mathematically intractable variance estimators.

While permutation tests and bootstraps have very wide-ranging application, both share a common potential drawback: as data-intensive resampling methods, both can be runtime prohibitive when applied to large or even medium-sized data samples drawn from large datasets. The data explosion over the past few decades has made this a common occurrence, and it highlights the increasing need for faster, and more efficient and scalable,[3] permutation test and bootstrap algorithms.

During the same time period as the rise of the bootstrap, SAS has grown to become one of the most widely used statistical computing platforms globally.[4] The footprint of researchers and analysts using SAS to perform both bootstraps and permutation tests, therefore, is considerable. This, and SAS's reputation for speed, make it a very good choice as a platform for developing fast, scalable bootstrap and permutation test algorithms, and probing and expanding the limits on the speed with which they can be executed on large datasets. Key to this effort, of course, is developing scalable, fast algorithms for simple random sampling with replacement (SRSWR) and simple random sampling without replacement (SRSWOR), which serve as the core of the bootstrap and permutation test algorithms, respectively.

This paper compares seven bootstrap and six permutation test algorithms coded in SAS. Among the permutation test algorithms, three (OPDN, OPDN-Alt, and Bebb-Sim) require only the Base SAS module, and three (PROCSS, PROCMT, and PROCNPAR) rely on SAS Procedures available as part of the SAS/STAT module. All implement the conventional sampling-without-replacement permutation test, where the null hypothesis is the equality of the two population distributions represented by the two samples, and the permuted statistic is the sample sum.[5] However, four of the algorithms (OPDN, OPDN-Alt, Bebb-Sim, and PROCSS) easily can be modified to permute any statistic, and the remaining two (PROCNPAR, which uses

[2] The only assumption made by all but very stylized permutation tests is that the subscripts of the data samples are exchangeable under the null hypothesis.

[3] An algorithm is "scalable" if it continues to function in reasonable timeframes when its inputs (here, the numbers of observations in the datasets or strata (*N*), and samples being tested (*n*), increase in scale. For example, if the sample sizes increase by a factor of ten, does the algorithm take ten times as long to complete? Only half as long? Or over one hundred times as long? Many consider an algorithm that increases by a factor of *n^2* or greater NOT to be "scalable." The OPDY and OPDN algorithms designed herein have sub-linear time complexity, that is, as *n* increases, the runtimes increase merely by approximately the square root of *n*, which generally is undeniably viewed as very "scalable."

[4] See www.SAS.com. SAS is arguably the largest privately owned software firm in the world, and with over 45,000 registered user sites, it is arguably the most widely used statistical platform globally, even without including users of the many SAS "clones" that exist as well (see WPS (from World Programming – see http://teamwpc.co.uk/home), Dap (see http://www.gnu.org/software/dap/ and http://en.wikipedia.org/wiki/DAP_(software)), and arguably Carolina (see DullesOpen.com, a unit of Dulles Research LLC), and formerly, PRODAS (from Conceptual Software) and BASS (from Bass Institute)).

[5] Proc Multtest, as used in PROCMT, uses pooled-variance t-statistics, which provide a permutation p-value mathematically identical to that provided by sample sums.

Proc NPAR1WAY, and PROCMT, which uses Proc Multtest) have a finite number of choices for permutation test statistics.

Among the bootstrap algorithms, five (OPDY, HTPS, HTHI, DA, Out-SM, and A4.8) require only the Base SAS module, and one (PSS) relies on a SAS Procedure (Proc SurveySelect) available as part of the SAS/STAT module. All implement the conventional sampling-with-replacement bootstrap.

Throughout this paper, N is the number of observations in the dataset or the stratum, and n is the size of the bootstrap or permutation test sample (and when m is presented, it represents the number of bootstrap or permutation test samples (of size n) being drawn).

Background: Exploiting the Fast Sequential Access of SAS

Key to this paper is the fact that SAS is not a "matrix" language (like MATLAB or Gauss) or a "vector" language (like S-Plus or R), but rather, with a few exceptions, it processes data sequentially, record-by-record. Since the late 1970's SAS has become extremely fast and efficient at such sequential record processing,[6] and naturally, this strongly shapes the algorithms presented herein. It also leads to a bit of a paradox when comparing the time complexity of these algorithms: the real runtime ranking of these algorithms, when implemented in SAS, can deviate notably from a ranking based on their theoretical time complexity. In other words, due to SAS's fast sequential access, O(N) algorithms are often "better" in SAS than O(n) algorithms, even when $N>>n$. While this is important to note, and an example is explained in the paper, the focus of this paper is real runtimes, and the speed with which SAS users can obtain actual results. Resource constraints such as I/O speeds, storage space, and memory are discussed below, and CPU runtimes are presented alongside real runtimes in the Results section, but the goal here is not to develop or compare algorithms based on their theoretical time complexity (although the time complexity of both OPDY and OPDN are presented and estimated, based on their empirical runtimes).

The Six Permutation Test Algorithms

The six algorithms include: OPDN ("One Pass, Duplicates? No"), OPDN-Alt ("One Pass, Duplicates-No - Alternate"), PROCSS (Proc SurveySelect), PROCMT (Proc Multtest), PROCNPAR (Proc NPAR1WAY), and Bebb-Sim (Simultaneous Bebbington). SAS v.9.2 code for all the algorithms is presented in Appendix A, along with code that generates the datasets on which the algorithms are run to produce the runtimes shown in the Results section. Each algorithm is a macro that is completely modular and takes input values for six macro variables: the name of the input dataset, the name of the output dataset, the name of the variable to be permuted, the "by variables" defining the strata, the name of the variable designating the "Control" and "Treatment" samples, the number of permutation samples to be used in the permutation test, and a value (optional) for a random number generation seed (one of the macros, PROCMT, takes a value for a seventh macro variable specifying upper or lower p-values). Aside from general SAS programming language rules (e.g. file naming conventions), assumptions made by the code are minimal (e.g. the variable designating the "Control" and "Treatment" samples contains corresponding values of "C" and "T", "by variables" are assumed to be character variables, and input datasets are presumed to be sorted by the specified "by variables" for all algorithms). Each algorithm is discussed below.

[6] To quote the author of a previous SAS® bootstrap paper that extols the virtues of PROC SurveySelect, "Tools like bootstrapping and simulation are very useful, and will run very quickly in SAS .. if we just write them in an efficient manner." (Cassell, 2007).

OPDN:
As first published in Opdyke (2011), this is a completely new and original memory-intensive algorithm, which is one of the reasons it is so fast. It uses no storage space, other than the original input dataset and a summary dataset of the record counts for each stratum. OPDN makes one pass through the dataset, record by record, stratum by stratum, and efficiently builds a large array of data values, essentially converting a column of data into a row of data (for one stratum at a time).[7] The fast sequential access of SAS ensures that the *N*-celled array (where *N* is the size of the current stratum) is loaded with data values automatically, and very quickly, simply by using a SET statement and a _TEMPORARY_ array (which automatically retains values across records as each record is being read). Then SRSWOR is performed, in memory, using this array. The approach for SRSWOR is essentially that of Goodman & Hedetniemi (1977) which, as defined by Tille (2006), is a draw-by-draw algorithm.[8] While Tille (2006) describes the standard draw-by-draw approach for SRSWOR as having the major drawback of being "quite complex" (p.47), the draw-by-draw SRSWOR algorithm used by OPDN unarguably is not: as shown below in pseudo code, it is straightforward and easily understood.

OPDN implementation #1 of Goodman & Hedetniemi (1977) for Permutation Tests:

*** temp[] is the array filled with all the data values, for the current stratum, of the variable being permuted
*** psums[] is the array containing the permutation sample statistic values for every permutation sample

```
do m = 1 to #permutation tests
   x ← 0
   tot_FREQ_hold ← # records in current stratum
   tot_FREQ ← tot_FREQ_hold
   do n = 1 to # records in smaller of Control and Treatment samples
      cell ← uniform random variate on 1 to tot_FREQ
      x    ← temp[cell] + x
      hold ← temp[cell]
      temp[cell] ← temp[tot_FREQ]
      temp[tot_FREQ] ← hold
      tot_FREQ ← tot_FREQ -1
   end;
   psums[m] ← x
end;
```

An explanation of the SRSWOR algorithm is as follows: random draws are made from the array using a uniform random variate, cell, drawn initially from 1 to *N* (where *N* is the size of the current stratum). The value from temp[cell] is used in the calculation of the sample statistic, and then swapped with that of temp[*N*]. This is repeated in a loop that iterates *n* times, where *n* is the size of the permutation sample (for efficiency, *n* is always the smaller of either the Control or Treatment sample, by design), but in each loop *N* is decremented by 1, so selected values cannot be selected more than once as they are placed at the end of the

[7] The "One Pass" referenced in the name of this algorithm refers to SAS® making one sequential pass through the dataset when reading it. The sampling without replacement that is done using the *N*-celled array (row), after the entire stratum of *N* data values has been read into the array, obviously is not sequential or "one-pass."

[8] "Definition 37. A sampling design of fixed sample size n is said to be draw by draw if, at each one of the n steps of the procedure, a unit is definitively selected in the sample" (p. 35).

array which is never again touched by the random number generator (see Appendix B for a simple proof of this algorithm as a valid SRSWOR algorithm).

An alternate presentation of the SRSWOR algorithm is shown below, and it utilizes the fact that even if statistical calculations are not being performed as the sample is being selected (as the sample sum is cumulatively calculated above), the entire without-replacement-sample ends up in the last *n* cells of the array. So if applying a function to this collective set of cells is faster or more efficient than cumulatively calculating the sample statistic, it would be the preferable approach.

OPDN implementation #2 of Goodman & Hedetniemi (1977) for Permutation Tests:

*** temp[] is the array filled with all the data values, for current stratum, of the variable being permuted
*** psums[] is the array containing the permutation sample statistic values for every permutation sample

```
do m = 1 to #permutation tests
   tot_FREQ_hold ← # records in current stratum
   tot_FREQ ← tot_FREQ_hold
   do n = 1 to # records in smaller of Control and Treatment samples
      cell ← uniform random variate on 1 to tot_FREQ
      hold ← temp[cell]
      temp[cell] ← temp[tot_FREQ]
      temp[tot_FREQ] ← hold
      tot_FREQ ← tot_FREQ -1
   end;
   psums[m] ← sum(temp[tot_FREQ] to temp[tot_FREQ_hold])
end;
```

While use of this general algorithm appears to be reasonably widespread (for example, see Pesarin, 2001, p.81), it does not appear to be common in some areas of statistics where it would be of benefit (for example, Tille (2006), which is an authoritative statistical sampling resource, does not appear to present it), nor does it appear to be in common usage within the SAS community.

In addition to its efficiency for drawing a single without-replacement sample, another crucially important advantage of using this particular SRSWOR algorithm for performing permutation tests, or any statistical procedure requiring many without-replacement samples, is that the same array of data values can be used repeatedly, for generating all the permutation (SRSWOR) samples, even though it has been reordered by the selection of the previous sample. Fortunately, the order of the data values in the array does not matter for SRSWOR: the selection of each item is random, so sampling on the array can be performed regardless of the initial order of the data values in its cells. This means that no additional storage space is required when generating all the permutation samples – for each new without-replacement sample, the array simply can be used as it was left from the previous sample. This makes the repeated implementation of SRSWOR, as required by permutation tests, extremely fast and efficient: its runtime complexity is $O(n)$, and its storage (memory) space complexity is $O(N)$.

Although Goodman & Hedetniemi's (1977) approach has been modified and improved to require even less than $O(N)$ storage space (see Ernvall & Nevalainen, 1982), SAS's very fast built-in sequential access automatically fills the entire *N*-celled (_TEMPORARY_) array much faster, under almost all conditions, than SAS could selectively fill an even much smaller array of size *n* by choosing specific records using, say, a point= option on a SET statement. With reasonable amounts of memory (e.g. 16GB of RAM), only if the

stratum size approaches one billion records will the *N*-cells be too many for a _TEMPORARY_ array and cause the program to crash (by comparison, Proc NPAR1WAY crashes on strata less than a tenth the size of those that OPDN can handle). So the OPDN code relying on Goodman & Hedetniemi's (1977) approach using the full *N*-celled (_TEMPORARY_) array is not only faster than alternatives with smaller theoretical time and space complexity, but also, in SAS, more robust than those alternatives, and more than adequate for handling sampling from all but the very largest of datasets.

A final point regarding OPDN: because it is memory intensive, OPDN uses code to optimize SAS's efficient use of memory, and to that end, the algorithm uses a data _null_ and saves the calculated permutation test results in macro variables cumulated by strata, rather than in a dataset specified in the data step (which reserves a nontrivial amount of memory and decreases the size of the strata OPDN could otherwise handle). This makes OPDN more robust, and for those familiar with SAS macro code, understanding it is not onerous. The precision loss due to saving numerical results in macro variables (which are character strings) is, for most applications, trivial – to the tenth decimal place rather than the fourteenth. If that additional precision is needed, the user probably should be using a symbolic programming language, such as Mathematica®.

As described below, OPDN-Alt handles the data-step memory conservation issue a bit differently and with slightly less code, but it typically is slightly slower as a result.

OPDN-Alt:
This algorithm (OPDN "Alternative") is essentially the same algorithm as OPDN but with one coding difference: instead of saving the permutation p-values in macro variables cumulated by strata, rather than in a dataset specified on the data step to conserve memory, the OPDN-Alt code creates an empty dataset with missing values for all the necessary variables, and then uses a modify statement to update those values.[9] Otherwise, the algorithm is identical. The only arguable advantage to this approach is that it uses slightly less code. The unarguable disadvantage, however, is that it typically is slightly slower. That it is almost as fast is not surprising, since it is essentially the same algorithm, with a slightly different (and less efficient) approach to handling memory constraints in SAS.

PROCSS:
PROCSS uses the built-in SAS procedure, Proc SurveySelect, to create a sampling variable in the original dataset indicating which records in the stratum are selected into the without-replacement random sample, and multiple without-replacement samples can be created automatically for multiple strata. After this sampling is performed by Proc SurveySelect, the (sometimes quite large) output sampling dataset is summarized according to the permutation statistic being used with a Proc Summary, and the subsequent code calculating the permutation test p-values is identical to that of OPDN and OPDN-Alt. According to the SAS Online Documentation, for simple random sampling without replacement, if there is enough memory Proc SurveySelect uses Floyd's ordered hash table algorithm (see Bentley and Floyd (1987) and Bentley and Knuth (1986) for details). If there is not enough memory available for Floyd's algorithm, Proc SurveySelect

[9] The idea of using a modify statement instead of macro variables cumulated by strata came from a SAS® user who emailed me to comment on the OPDY bootstrap algorithm presented in Opdyke (2010). When I asked for his/her name, he/she would not provide it, so I can only give credit indirectly for the suggestion to try this approach when implementing the OPDN algorithm presented herein. I also modified the OPDY bootstrap algorithm presented in Opdyke (2010) and implemented it as OPDY-Alt, with very similar results: OPDY is slightly faster than OPDY-Alt.

switches to the sequential algorithm of Fan, Muller, and Rezucha (1962), which requires less memory but might require more time to select the sample.[10]

Aside from speed, which is discussed in the Results section, the major disadvantage of relying on Proc SurveySelect instead of OPDN, OPDN-Alt, or Bebb-Sim (see below) is that it requires disk space for the sometimes very large output sampling dataset that it generates, while these three other algorithms have no such resource requirement.

PROCMT:
PROCMT uses Proc Multtest to calculate permutation test p-values, and prior to the relatively recent advent of Proc SurveySelect, Proc Multtest had been widely used for this purpose. Although Proc Multtest does not explicitly allow the user to specify sample sums as the permutation test statistic, using t-scores based on the pooled-variance provides mathematically identical permutation test p-values, so the pooled-variance option is specified in PROCMT. This option is easily changed if the assumption is not warranted by the dataset in use.

One limitation of Proc Multtest is that for each time the Proc is used, it only provides one p-value (either the left, right, or two-tailed p-value). So to obtain all three p-values the user must run the Proc three times. For comparison purposes in this paper, PROCMT runs Proc Multtest twice, obtaining one one-tailed p-value (specified by the user in a macro variable) and the two-tailed p-value. Since Proc NPAR1WAY provides the smaller of the two one-tailed p-values and the two-tailed p-value, and OPDN, OPDN-Alt, Bebb-Sim, and PROCSS provide all three p-values, this appeared to be a reasonable compromise for purposes of comparison. Obviously, if the user knows in advance that he/she only needs one of the three p-values, then the runtimes presented in the Results section for PROCMT will be slightly less than twice as large. Note, however, that none of the p-values can be calculated based on the other p-values: each must be generated independently.

Aside from speed, which is discussed in the Results section, the major disadvantage of relying on Proc Multtest instead of OPDN, OPDN-Alt, Bebb-Sim, or PROCSS is that the range of permutation test statistics available to the user is limited, whereas any of the latter can use any permutation test statistic.

PROCNPAR:
PROCNPAR uses Proc NPAR1WAY to generate the smaller of the two one-tailed permutation test p-values and the two-tailed permutation test p-value. The options specified execute Pitman's permutation test, which uses the data values themselves as scores, and the sum as the sample statistic, consistent with the five other algorithms presented herein. The monte carlo option is used so that the number of permutation samples can be specified by the user.

Aside from speed, which is discussed in the Results section, there are two major disadvantages to relying on Proc NPAR1WAY instead of OPDN. First, for a fixed amount of memory, Proc NPAR1WAY crashes on strata less than a tenth the size of those that OPDN can handle. Secondly, like Proc Multtest, the range of permutation test statistics available to the user is limited, whereas OPDN, OPDN-Alt, Bebb-Sim, and PROCSS can use any permutation test statistic.

[10] Note that the SASFILE statement used with Proc SurveySelect (see Cassell, 2007) is useless when the datasets to be permuted are too large for the extant memory – and that is the only time that fast permutation tests really are needed.

Bebb-Sim:
"Simultaneous Bebbington" refers to Bebbington (1975), one of the first and most straightforward sequential-sampling-without-replacement algorithms. Like the SRSWOR used in OPDN, Bebbington requires that N is known ahead of time, and it makes exactly n selections from N items, each with equal probability. But unlike OPDN, Bebbington is sequential: it makes a sample selection decision for each data record as it is encountered, sequentially (in order), in the dataset. Bebbington (1975) is presented in pseudo-code below for the reader's convenience.

```
1. Initialize: Let i ← 0, N′ ← N + 1, n′ ← n
2. i ← i + 1
3. If n′ = 0, STOP Algorithm
4. Visit Data Record i
5. N′ ← N′ – 1
6. Generate Uniform Random Variate u ~ Uniform[0, 1]
7. If u > (n′ / N′), Go To 2.
8. Otherwise, Output Record i into Sample
9.            n′ ← n′ – 1
10.          Go To 2.
```

Bebbington's (1975) algorithm above guarantees that 1) all possible samples of size n drawn from N are equally likely; 2) exactly n items will be selected; 3) no items in the sample will be repeated; and 4) items in the sample will appear in the same order that they appear in the population dataset.

The "Simultaneous" refers to the fact that two arrays of size m (where m is the number of permutation samples) are created and used in the dataset to simultaneously execute Bebbington m times as the dataset is being (sequentially) read, record by record. One array contains the cumulated test statistics for that particular sample, and the other array contains the corresponding counters for each sample, counting the number of items already selected into that sample (n' in the algorithm shown above). The counters are necessary to apply the correct probability of selection to each item in order to make the algorithm a valid SRSWOR procedure.

Aside from speed, which is discussed in the Results section, an advantage of relying on Bebb-Sim rather than Proc SurveySelect is that it is memory intensive, like OPDN, and does not require any disk space beyond the input dataset and a small dataset of counts for each stratum.

Other Possibilities:

Proc Plan:
Proc Plan can be used to conduct permutation tests, but since it does not have "by statement" functionality, it must be used once for each set of by-variable values in the original dataset. These results can be SET together and merged with the original dataset (after an observation counter that counts within each stratum is created) to obtain without-replacement samples. However, this is a much, much slower method, and possibly part of the impetus for the more recent creation of Proc SurveySelect.

DA:
The widely used and aging DA (Direct Access) method, which uses a SET statement with a point=x option (where x is a uniform random variate) to randomly select with equal probability an observation in a dataset, was shown by Opdyke (2010) to be far less efficient and far more than an order of magnitude slower than alternatives such as the OPDY algorithm for calculating bootstraps (see Opdyke, 2010). In the permutation test setting, DA is now actually irrelevant, since the only way (known to this author) to implement it is using Bebbington (1975) in a nested loop, where an inner loop iterates *n* times to obtain a single sample without replacement, and an outer loop iterates *m* times to obtain *m* without-replacement permutation samples (using this approach with only the inner loop to obtain a single SRSWOR sample is common to many SAS programs). While technically possible to implement, a nested loop is extremely slow for this approach, and so it is not viable as a scalable algorithm for executing fast permutation tests in SAS.

Results of Permutation Test Algorithms

The real and CPU runtimes of each of the algorithms, relative to those of OPDN, are shown in Table 1 below for different #strata, *N* = strata size, and *m* = size of the permutation samples (*n* = permutation sample size = 1,000 for all runs; for the absolute runtimes, please contact the author). The code was run on a Windows PC with only 2 GB of RAM and a 2 GHz Pentium chip.

OPDN dominates in all cases, except that typically it is only slightly faster than OPDN-Alt which, as mentioned above, is essentially the same algorithm, but with a slightly different and less efficient approach to memory conservation.

The second fastest algorithm is Bebb-Sim, followed by PROCNPAR for the smaller datasets but PROCSS for the larger datasets, and then PROCMT is much slower for all but the smallest datasets. Generally, the larger the dataset, the larger the runtime premium OPDN has over the other algorithms, achieving real runtime speeds 218x faster than PROCSS, 353x faster than PROCNPAR, and 723x faster than PROCMT. If it was runtime feasible to run PROCSS and PROCMT on even larger datasets, all indications are that the runtime premium of OPDN would only continue to increase. This would not be possible for PROCNPAR, however, because Proc NPAR1WAY crashes on datasets less than a tenth the size of those OPDN can handle (with only 2 GB of RAM, NPAR1WAY crashed on less than 10 million observations in the largest stratum, while OPDN handled over 110 million observations in the largest stratum).

Note that another advantage OPDN has over PROCSS (Proc SurveySelect) is that it needs virtually no storage (disk) space beyond the input dataset, while PROCSS needs disk space to output a potentially very large sampling dataset that subsequently must be summarized at the sample level, according to the permutation test statistic being used, to compare the permuted sample statistics to the original sample statistic.

Table 1: Real and CPU Runtimes of the Algorithms Relative to OPDN for Various *N*, #strata, and *m*
(**EX** = excessive, **CR** = crashed, sample size *n* = 1,000 for all)

			REAL					CPU				
N ***(per stratum)***	***#*** ***strata***	***m***	**OPDN -Alt**	**PROC SS**	**PROC MT**	**PROCN PAR**	**Bebb -Sim**	**OPDN -Alt**	**PROC SS**	**PROC MT**	**PROC NPAR**	**Bebb -Sim**
10,000	2	500	1.2	11.1	4.7	4.2	2.9	1.1	9.1	6.0	4.1	4.2
100,000	2	500	1.7	19.8	66.9	15.9	15.4	1.1	21.7	96.7	22.3	22.3
1,000,000	2	500	1.1	33.7	177.3	47.2	29.4	1.1	46.7	321.6	85.3	53.4
10,000,000	2	500	1.0	44.6	255.5	CR	36.4	1.0	57.4	422.1	CR	60.2
10,000	6	500	1.3	11.6	6.2	4.9	4.2	1.0	9.1	6.5	4.0	4.4
100,000	6	500	1.2	25.5	85.7	20.2	19.7	1.0	21.0	95.1	22.1	21.8
1,000,000	6	500	1.1	30.6	160.5	43.5	26.7	1.1	43.7	307.8	82.6	51.3
10,000,000	6	500	1.0	45.1	EX	CR	40.3	1.1	49.5	EX	CR	59.1
10,000	12	500	1.5	12.2	5.9	4.0	4.2	1.0	9.2	6.5	4.1	4.7
100,000	12	500	1.2	28.4	88.0	21.7	21.1	1.0	23.1	94.7	23.0	22.8
1,000,000	12	500	1.1	41.6	222.1	61.8	37.4	1.1	41.4	293.6	81.3	49.4
10,000,000	12	500	1.1	53.3	EX	CR	45.1	1.1	54.1	EX	CR	59.2
10,000	2	1000	1.1	15.2	6.5	4.0	4.4	1.0	10.3	6.8	4.1	4.7
100,000	2	1000	1.1	33.6	104.9	23.7	24.3	1.0	29.2	115.7	26.0	26.6
1,000,000	2	1000	1.3	96.6	468.9	125.0	79.1	1.0	82.3	508.0	134.7	85.8
10,000,000	2	1000	1.0	80.6	504.3	CR	73.3	1.0	96.6	794.0	CR	115.5
10,000	6	1000	1.0	12.6	6.5	4.0	4.6	1.0	9.5	6.8	4.0	4.8
100,000	6	1000	1.1	34.6	113.3	25.8	26.5	1.0	27.6	120.3	27.1	28.1
1,000,000	6	1000	1.0	55.7	291.8	77.6	50.1	1.1	74.9	520.4	138.0	89.2
10,000,000	6	1000	1.0	85.3	EX	CR	79.2	1.1	94.3	EX	CR	115.4
10,000	12	1000	1.0	13.3	6.6	4.3	4.7	1.0	9.6	6.7	4.2	4.8
100,000	12	1000	1.0	33.4	115.0	27.4	26.5	1.0	25.8	117.6	27.7	27.1
1,000,000	12	1000	1.0	78.4	412.4	111.6	70.1	1.1	74.1	516.1	139.3	87.8
10,000,000	12	1000	1.0	99.8	EX	CR	88.7	1.1	91.3	EX	CR	111.5
10,000	2	2000	1.3	10.9	5.4	3.5	3.9	1.0	9.3	6.5	3.9	4.7
100,000	2	2000	1.0	36.2	122.5	27.8	28.3	1.0	29.6	132.6	29.7	30.6
1,000,000	2	2000	1.0	135.9	723.1	191.2	123.1	1.0	116.1	798.5	211.2	136.1
10,000,000	2	2000	1.0	172.7	EX	CR	143.6	1.0	215.8	EX	CR	227.9
10,000	6	2000	1.0	13.6	6.5	4.0	4.6	1.0	10.3	6.7	4.0	4.7
100,000	6	2000	1.0	38.8	126.1	28.6	29.5	1.0	31.0	131.5	29.7	30.7
1,000,000	6	2000	1.0	97.7	497.9	136.4	85.5	1.0	122.8	810.3	214.8	138.7
10,000,000	6	2000	1.1	218.2	EX	CR	159.0	1.1	248.3	EX	CR	222.9
10,000	12	2000	1.0	15.0	6.5	4.2	4.7	1.0	11.5	6.7	4.2	4.8
100,000	12	2000	1.0	44.4	131.9	30.7	30.4	1.0	35.5	135.4	31.5	31.3
1,000,000	12	2000	1.0	147.0	685.1	185.7	117.2	1.1	136.4	798.6	216.4	136.5
10,000,000	12	2000	1.1	EX	EX	CR	EX	1.1	EX	EX	CR	EX
7,500,000	2	2000	1.1	243.6	EX	353.0	201.0	1.1	214.0	EX	396.3	227.5
7,500,000	6	2000	1.2	242.0	EX	334.8	193.9	1.1	218.2	EX	382.5	222.9
25,000,000	12	500	1.2	EX	EX	CR	EX	1.1	EX	EX	CR	EX
50,000,000	12	500	1.2	EX	EX	CR	EX	1.1	EX	EX	CR	EX
100,000,000	12	500	1.0	EX	EX	CR	EX	1.0	EX	EX	CR	EX

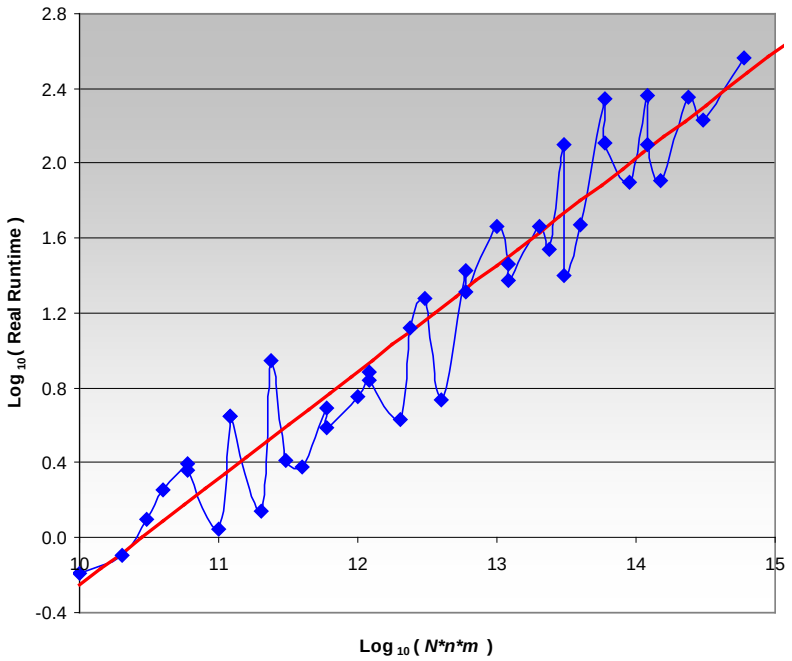
And finally, as mentioned above, the size constraint on OPDN is the size of the largest stratum in the dataset, *not* the size of the entire dataset, which appears to be what drives the runtimes of Proc SurveySelect and Proc Multtest (this does not appear to be the case for Proc NPAR1WAY, but then, Proc NPAR1WAY crashes on much smaller strata). So OPDN maintains scalability on datasets potentially many, many orders of magnitude larger than those the three Proc's can handle.

In a nutshell, OPDN is not only much, much faster than all the other methods, but also far more robust. All of these results combine to make OPDN the only truly scalable permutation test (and SRSWOR) algorithm in SAS.

Focusing on the real runtime of OPDN and its relationship to #strata, *N*, *n*, and *m* (see Figure 1), the empirical runtimes (available from the author upon request) yield a reasonably accurate approximation in (1). So given the 0.57 (≈ 0.50) term in (1) below, OPDN appears to be approximately O($\sqrt{N \times n \times m}$), which makes sense given its composition. And sub-linear time complexity is a highly desirable algorithmic property from the perspective of scalability, one which the relevant Proc's do not share.

$\text{Log}_{10}(\text{Real Runtime}) = -5.95291 + 0.57001 * \text{Log}_{10}(N*n*m)$ (where *N* = all *N* across strata) (1)

Figure 1: OPDN Real Runtime by *N*n*m* (*N* = all strata)



The Seven Bootstrap Algorithms

The seven algorithms compared in this paper include: OPDY ("One-Pass, Duplicates?-Yes"), Proc SurveySelect (PSS), HTPS (Hash Table, Proc Summary), HTHI (Hash Table, Hash Iterator), Direct Access (DA), Output-Sort-Merge (Out-SM), and Algorithm 4.8 (A4.8). SAS v.9.2 code for all the algorithms except the two hash algorithms is presented in Appendix C. All except OPDY and A4.8 have been widely used in the SAS® literature, if not specifically for bootstraps, then for very closely related analyses. Each is

completely modular and takes only five macro variables as inputs: the size of the bootstrap samples (*n* observations), the number of bootstrap samples (*m*), the "by variables" defining the strata, the input dataset, the variable to be bootstrapped, and the output dataset. All are described briefly below.

OPDY:
As first published in Opdyke (2010), this is a completely new and original memory-intensive algorithm, which is why it is so fast. It uses no storage space, other than the original input dataset and a summary dataset of the record counts for each stratum. OPDY makes one pass through the dataset, record by record, stratum by stratum, and efficiently builds a large array of data values, essentially converting a column of data into a row of data. Then random draws are made from the array using a uniform random variate, *u*, drawn from 1 to *N* (where *N* is the size of the current stratum).[11] Bootstrap results from each stratum are saved in cumulated macro variables to minimize the intermediate memory requirements of outputting a dataset in the data step. The only constraint on this algorithm is *not* the total number of records across all strata but rather, the number of records in the largest stratum, which determines the size of the array held in memory. Yet even on a system with very little memory (2 gigabytes of RAM), the algorithm is able to process input datasets where the largest stratum contains over 110 million records, so this should not be a constraint for most users and most practical uses (the hashing algorithms described below sometimes crash with a largest stratum size of only about three to four million records, in datasets with only two "by variables" defining twelve strata). With more reasonable memory resources (e.g. 16 gigabytes of RAM), OPDY can process datasets where the largest strata approach a billion records.

Note that OPDY is easily generalizable to many multivariate models. For example, say five covariates are being regressed on one dependent variable in an ordinary least squares (OLS) bootstrap (of the entire observation, not the residuals). This would simply require that a separate *N*-celled array be loaded for each covariate and the dependent variable. The same *u* is then used to draw once from each of the *N*-celled arrays, and this is done, with replacement, *n* times to obtain one bootstrap sample. Then $(X'X)^{-1}X'Y$ is calculated using the six *n*-celled arrays, and the result – a vector of five parameter estimates – is saved. This is done *m* times to obtain *m* bootstraps of the five parameter estimates. The maximum number of observations per strata would be one sixth what it would be for a univariate bootstrap.

PSS:
PSS uses the SAS® procedure Proc SurveySelect to create a sampling variable in the original dataset indicating how many times record *i* is selected into the with-replacement random sample, and multiple bootstrap samples can be created automatically for multiple strata. These are then summarized by another procedure to obtain the desired bootstrap statistics.[12]

DA:
DA is merely code for Directly Accessing specified records within a SAS® dataset. A uniform random variate, *u*, is drawn from 1 to *N* (the size of the stratum), and record # *u* is selected from the dataset using the point= option on the set statement. In a nested loop, this is done *n* times to obtain a single bootstrap sample,

[11] The "One Pass" referenced in the name of this algorithm refers to SAS® making one sequential pass through the dataset when reading it. The sampling with replacement that is done using the *N*-celled array, after the entire stratum of *N* data values has been read into the array, obviously is not sequential or "one-pass."

[12] Note that PROC PLAN was not tested because it does not sample with replacement, and PROC MultTest was not tested because it conducts only *n*-out-of-*n* bootstraps and does not allow the user to define the size of the bootstrap sample (aside from changing the size of the input dataset, which would require an extra randomization step which would defeat the purpose of conducting efficient sampling with replacement). And the SASFILE statement used with PROC SurveySelect (see Cassell, 2007) is useless when the datasets to be bootstrapped are too large for the extant memory – and that is the only time that fast bootstraps really are needed.

and then the entire process is repeated *m* times to obtain *m* bootstrap samples. The bootstrap samples do not need to be summarized after the data step because the bootstrap statistics are created sequentially, as each bootstrap sample is created, saved in an array, and then summarized by functions applied to the array when the end of the stratum is reached, all within the data step. DA requires virtually no storage space beyond the original input dataset.

Out-SM:
Out-SM also uses a uniform random variate, *u*, drawn from 1 to *N* to identify the records to include in the samples, but instead of using direct access it creates a new dataset by outputting, in a nested loop, one record for each record # identified by *u*, looping *n* times to create one bootstrap sample. This is done *m* times in a nested loop to create *m* bootstrap samples, while keeping track of both the record numbers (*u*) and the number of the bootstrap sample (*m*). This new dataset is then sorted and merged with the original dataset (in which a record counter was created) to obtain the actual data records for each bootstrap sample. These are then summarized at the level of the bootstrap sample.

HTHI:
The Hash code used in HTHI follows two published SAS® papers (see below). The structure of the code is essentially the same as that of Out-SM, except that a hash table, instead of a merge statement, is used to merge the original dataset and the bootstrap sample dataset (see Secosky and Bloom, 2007), and a hash iterator, instead of Proc Summary, is used to summarize the bootstrap samples (see Secosky and Bloom, 2008). This algorithm is memory-intensive, as well as being storage-intensive, but it is fast. However, it has two major drawbacks: first, the hash iterator can only sum the values of the variable being bootstrapped, so if the statistic of interest requires more complex calculations, additional or separate data steps and/or procedures would be required (this is handled by HTPS, which uses a Proc Summary instead, but which is slightly slower as a result). Secondly, the memory constraints of hashing are far greater than those of OPDY, causing it to crash under dataset sizes much smaller than those OPDY can handle (this constraint also applies to HTPS below). For example, using input datasets with two "by variables" defining twelve strata, OPDY can handle strata with over 110 million records, while both hashing algorithms often crash when only three to four million records per strata are used. And of course, since the size constraint on OPDY is the number of records in the largest stratum and *not* the number of records overall in the dataset, it actually can handle datasets orders of magnitude larger than those the hashing algorithms can handle, as long every stratum is below a certain size (in this study, about 110 million records).

HTPS:
This algorithm is identical to HTHI except that Proc Summary is used instead of the hash iterator. While slower than the hash iterator, Proc Summary allows the user to bootstrap any statistic, not just those based on the sum of the variable of interest.

A4.8:
I believed I had discovered this with-replacement sampling algorithm, but it is, in fact, listed as Algorithm 4.8 in Tillé (2006), who calls it "neglected." The original source of the algorithm is unknown to Tillé (see email correspondence, 09/30/10), and no proof of its validity is provided in Tillé (2006), so a proof is provided herein in Appendix D. The main advantage of this algorithm is that it requires virtually no storage space beyond the original input dataset (no extra bootstrap "sample" dataset is created, as in Out-SM and the two hashing algorithms), and it has relatively low memory requirements. In addition, no nested loops are required, like the *n* x *m* nested loops required for DA, Out-SM, and the hashing algorithms: A4.8 requires only a single pass through the dataset, with one loop performed *m* times on each record, to generate the desired statistic for all *m* bootstrap samples.

Although not noted elsewhere, the algorithm closely follows the structure of Bebbington (1975), one of the first and most straightforward sequential-sampling-without-replacement algorithms. Like Bebbington (1975), the algorithm is sequential, it requires that N is known ahead of time, and it makes exactly n selections from N items, each with equal probability. It makes one important modification to Bebbington (1975) that transforms it from a without-replacement algorithm to a with-replacement algorithm: it uses binomial random variates instead of uniform random variates. Below, for clarity, both Bebbington and A4.8 are presented.

1. Initialize: Let $i \leftarrow 0$, $N' \leftarrow N + 1$, $n' \leftarrow n$
2. $i \leftarrow i + 1$
3. If $n' = 0$, STOP Algorithm
4. Visit Data Record i
5. $N' \leftarrow N' - 1$
6. Generate Uniform Random Variate $u \sim$ Uniform[0, 1]
7. If $u > n' / N'$, Go To 2.
   Otherwise, Output Record i into Sample
   $n' \leftarrow n' - 1$
   Go To 2.

Bebbington's (1975) algorithm above guarantees that 1) all possible samples of size n drawn from N are equally likely; 2) exactly n items will be selected; 3) no items in the sample will be repeated; and 4) items in the sample will appear in the same order that they appear in the population dataset. The A4.8 algorithm presented below guarantees 1), 2), and 4), and instead of 3), it allows duplicates into the sample.

1. Initialize: Let $i \leftarrow 0$, $N' \leftarrow N + 1$, $n' \leftarrow n$
2. $i \leftarrow i + 1$
3. If $n' = 0$, STOP Algorithm
4. Visit Data Record i
5. $N' \leftarrow N' - 1$
6. Generate Binomial Random Variate $b \sim$ Binomial(n', $p \leftarrow 1/N'$) [13]
7. If $b = 0$, Go To 2.
   Otherwise, Output Record i into Sample b times
   $n' \leftarrow n' - b$
   Go To 2.

The only difference between the code implementation of A4.8 in Appendix C and its algorithm definition above is that the check of $n' = 0$ on line 3. is excluded from the code: execution simply stops when all records in the dataset (stratum) are read once through, sequentially. This is more efficient for these purposes, since all bootstrap samples are being created simultaneously, and the likelihood that all of them will be complete before that last record of the dataset (stratum) is read is usually very small – not nearly large enough to justify explicitly checking for it on line 3.

[13] Using A4.8, p will never equal zero. If $p = 1$ (meaning the end of the stratum (dataset) is reached and $i = N$, $N' = 1$, and $n' = 0$) before all n items are sampled, the rand function b=rand('binomial',p,n') in SAS® assigns a value of n' to b, which is correct for A4.8.

Results of Bootstrap Algorithms

The real and CPU runtimes of each of the algorithms, relative to those of OPDY, are shown in Table 2 below for different #strata, *N*, *n*, and *m* (for the absolute runtimes, please contact the author). The code was run on a PC with 2 GB of RAM and a 2 GHz Pentium chip. OPDY dominates in all cases that matter, that is, in

Table 2: Real and CPU Runtimes of the Algorithms Relative to OPDY for Various *N*, #strata, *n*, & *m*
(**EX** = excessive, **CR** = crashed, n=m)

			REAL						CPU					
N	**#**							**Out-**						**Out-**
(per stratum)	***strata***	*n=m*	**PSS**	**A4.8**	**HTPS**	**HTIT**	**DA**	**SM**	**PSS**	**A4.8**	**HTPS**	**HTIT**	**DA**	**SM**
10,000	2	500	5.3	10.2	9.0	4.5	8.8	7.5	2.6	10.1	3.3	2.1	3.8	6.0
100,000	2	500	23.3	83.3	6.2	4.6	7.3	9.7	18.0	81.4	7.3	2.4	3.1	4.4
1,000,000	2	500	31.1	121.2	2.5	1.9	1.4	4.2	25.0	119.3	1.2	1.2	0.7	1.8
10,000,000	2	500	42.2	164.9	2.5	2.7	0.7	7.6	33.4	162.0	1.4	1.4	0.4	1.8
10,000	6	500	8.4	16.8	10.0	6.8	13.8	14.3	5.8	16.6	4.5	3.6	6.0	7.6
100,000	6	500	23.7	84.0	4.7	4.3	7.4	8.7	18.9	82.7	2.8	2.5	3.3	4.4
1,000,000	6	500	37.3	145.0	2.3	2.1	4.4	4.5	30.4	143.2	1.4	1.4	0.8	2.2
10,000,000	6	500	39.8	370.8	**CR**	**CR**	7.8	8.8	32.5	349.1	**CR**	**CR**	0.4	1.8
10,000	12	500	11.4	23.7	17.9	28.0	18.1	16.8	8.5	23.1	5.8	4.6	8.0	9.6
100,000	12	500	31.4	96.3	9.4	9.0	8.7	8.6	24.7	94.2	3.2	2.7	3.4	5.0
1,000,000	12	500	47.0	160.7	3.8	3.3	10.1	4.2	38.1	155.8	1.7	1.6	0.9	2.2
10,000,000	12	500	45.0	**EX**	**CR**	**CR**	11.0	7.1	36.8	**EX**	**CR**	**CR**	0.4	1.8
10,000	2	1000	5.3	7.3	7.2	5.7	11.2	12.4	3.6	7.0	3.7	2.8	5.3	7.2
100,000	2	1000	25.7	80.4	8.5	7.4	14.3	16.0	19.9	78.1	4.7	3.6	6.4	8.4
1,000,000	2	1000	50.0	183.9	3.3	2.5	8.9	4.4	39.9	179.7	1.9	1.6	1.6	2.7
10,000,000	2	1000	65.2	1428.7	2.5	2.3	9.1	6.2	52.1	1218.7	1.2	1.1	0.5	1.6
10,000	6	1000	8.2	10.7	18.3	17.9	16.9	26.8	5.3	10.4	5.4	4.3	7.6	10.7
100,000	6	1000	28.3	87.8	14.5	13.6	15.6	25.5	21.5	86.4	5.3	4.4	6.3	9.5
1,000,000	6	1000	56.8	214.4	5.2	4.8	4.2	8.5	46.0	211.3	2.2	2.0	1.7	3.2
10,000,000	6	1000	63.6	**EX**	**CR**	**CR**	9.5	6.9	51.0	**EX**	**CR**	**CR**	0.5	1.5
10,000	12	1000	11.2	15.8	19.6	18.0	24.3	37.8	8.0	15.3	8.4	6.5	10.7	16.4
100,000	12	1000	15.1	47.7	6.1	6.0	8.7	12.2	11.8	47.2	2.9	2.2	3.6	5.3
1,000,000	12	1000	61.1	213.6	4.4	4.8	10.1	8.8	50.4	210.6	2.3	2.0	1.8	3.3
10,000,000	12	1000	63.1	**EX**	**CR**	**CR**	10.6	6.0	51.7	**EX**	**CR**	**CR**	0.5	1.4
10,000	2	2000	10.6	8.8	27.0	25.6	28.6	49.2	7.6	8.6	9.3	7.1	12.4	22.2
100,000	2	2000	14.2	39.1	12.5	14.8	14.4	20.8	10.8	38.2	4.5	3.5	5.8	10.5
1,000,000	2	2000	45.6	167.9	5.3	5.6	6.5	11.6	36.5	166.2	2.6	2.1	2.6	4.8
10,000,000	2	2000	87.2	**EX**	3.4	2.8	8.5	6.1	69.4	**EX**	1.3	1.1	0.7	1.8
10,000	6	2000	13.5	10.0	29.6	20.8	32.0	92.9	8.5	9.8	11.0	8.1	13.4	26.1
100,000	6	2000	23.9	63.7	19.5	17.0	23.1	50.5	18.0	62.8	7.4	5.9	8.9	17.6
1,000,000	6	2000	62.6	230.3	7.4	6.6	8.8	32.1	50.0	224.2	3.6	3.0	3.3	7.0
10,000,000	6	2000	85.9	**EX**	**CR**	**CR**	7.8	12.7	68.2	**EX**	**CR**	**CR**	0.7	1.8
10,000	12	2000	13.1	10.2	27.1	24.3	33.1	85.6	9.1	10.0	11.6	8.6	14.0	28.1
100,000	12	2000	26.4	63.5	17.3	15.9	25.4	55.4	18.5	62.5	8.2	6.2	10.1	19.0
1,000,000	12	2000	62.2	207.9	7.0	6.6	8.1	21.0	49.7	205.2	3.6	2.9	3.0	6.2
10,000,000	12	2000	50.2	**EX**	**CR**	**CR**	6.0	3.8	40.5	**EX**	**CR**	**CR**	0.4	1.0

all cases where the absolute runtimes are not trivial (i.e. under one minute).[14] This runtime efficiency gap grows dramatically in absolute terms for OPDY vs. most of the other algorithms as #strata, *N*, *n*, and/or *m* increase (to take just one example, when the input dataset contains 6 strata with only *N* =1,000,000 records each, and *n* = *m* = 2,000, the real runtime of OPDY is 36 seconds, compared to real runtimes of almost 38 minutes (PSS), about 2 hours, 19 minutes (A4.8), almost 5 minutes (HTPS), almost 4 minutes (HTPI), almost five and a half minutes (DA), and over 19 minutes (Out-SM). That OPDY so convincingly beats DA is mildly surprising given how widely known, how used, and how old DA is. Similarly, PSS, A4.8, and Out-SM never are serious competitors against OPDY. And even the algorithms that keep up better with OPDY, namely the two hashing algorithms, crash under dataset sizes that OPDY handles easily. As mentioned above, using input datasets with two "by variables" defining twelve strata, OPDY can handle multiple strata with slightly over 110 million records each, while both hashing algorithms often crash when the input dataset contains only three to four million records per strata. And of course, since the size constraint on OPDY is the number of records in the largest stratum and *not* the number of records overall in the dataset, it can actually handle datasets orders of magnitude larger than those the hashing algorithms can handle, as long every stratum is below a fixed size (here, about 110 million records). Obviously memory constraints for the hashing algorithms can be avoided with more memory, but it is the *relative* performance for a fixed amount of memory that matters, and OPDY's more efficient use of memory clearly trumps that of the hashing algorithms.

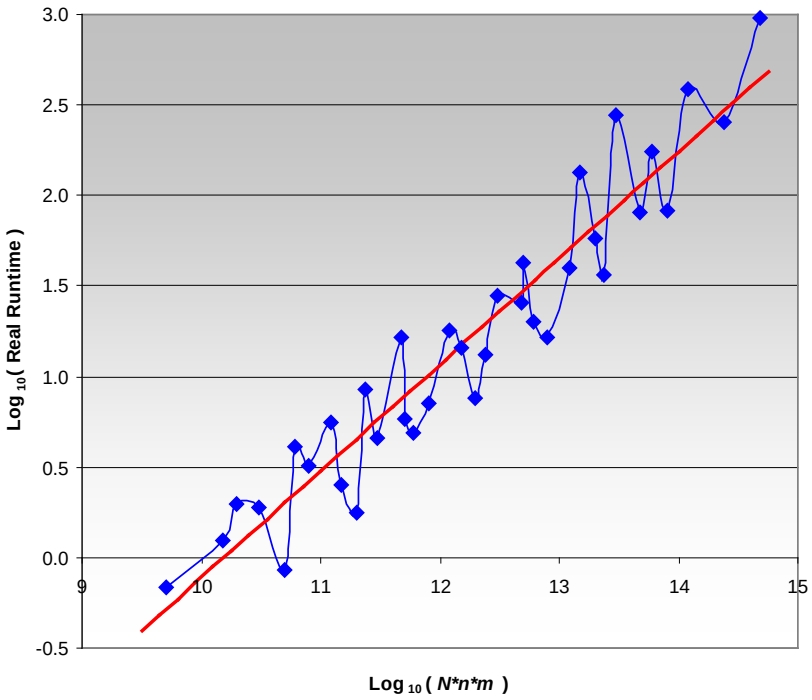
Aside from speed, note, too, that PSS, the two hashing algorithms, and Out-SM have the added disadvantage of requiring storage space for the bootstrap sample datasets, which can become prohibitively large if #strata, *N*, *n*, and/or *m* are large. All of these results combined arguably make OPDY the only truly scalable bootstrap algorithm in SAS.

Focusing on the real runtime of OPDY and its relationship to #strata, *N*, *n*, and *m* (see Figure 2), the empirical runtimes (available from the author upon request) yield a reasonably accurate approximation in (2). So given the 0.58 (≈ 0.50) term in (2), OPDY appears to be approximately $O(\sqrt{N \times n \times m})$, which makes sense given its composition. And sub-linear time complexity is a highly desirable algorithmic property from the perspective of scalability, one which Proc SurveySelect and the hashing algorithms do not share.

[14] In the only case where any of the algorithms is faster than OPDY (relatively few records (#strata = 2, *N* = 10,000,000) and relatively few bootstrap samples required (*n* = *m* = 500)), DA is faster by less than 15 seconds of real runtime. In absolute terms, however, total runtimes for both OPDY and DA are under a minute, making this irrelevant to solving the problem of developing fast, scalable bootstraps. DA fails on this front when absolute runtimes are large enough to actually matter. To take just one example, when the input dataset contains 12 strata with only 100,000 records each, but 2,000 bootstrap samples of size *n* = 2,000 are drawn, OPDY runs in under 26 seconds real time, while DA runs in about 11 minutes real time; if hundreds of such datasets require the application of bootstraps, DA becomes runtime prohibitive and is excluded as a viable option.

$\text{Log}_{10}(\text{Real Runtime}) = -5.99228 + 0.588164 * \text{Log}_{10}(N*n*m)$ (where N = all N across strata) (2)

Figure 2: OPDY Real Runtime by *N*n*m* (*N* = all strata)



Conclusions

The goal of this paper was to develop non-resource intensive, with- and without-replacement sampling algorithms to execute bootstraps and permutation tests faster than any competitors on arguably the most widely used statistical software platform globally. The OPDY and OPDN algorithms accomplish this objective. For bootstraps, OPDY uses negligible storage space, and on the SAS platform, it is much faster than any other alternative, including the built-in SAS Procedure (Proc SurveySelect) designed specifically for this purpose. It is even much faster than hashing, and simultaneously does not have the large storage requirements of hashing. Most importantly, the hashing algorithms crash under dataset sizes much smaller than those which OPDY can handle. Finally, OPDY's time complexity (approximately $O(\sqrt{N \times n \times m})$) is sub-linear, making it the only truly scaleable bootstrap algorithm in SAS®.

For permutation tests, OPDN uses negligible storage space, relatively little memory space via very efficient SRSWOR, and on the SAS platform, it is much faster than any other alternative, including the built-in SAS/STAT Procedures designed to perform permutation tests (Proc SurveySelect, Proc Multtest, and Proc NPAR1WAY). OPDN's relative (and absolute) speed premium increases with dataset size, and with reasonable, sub-linear time complexity (approximately $O(\sqrt{N \times n \times m})$) it maintains scalability on large datasets, handling datasets (technically, the largest stratum in the dataset) more than ten times larger than those that crash Proc NPAR1WAY. So it is not only the fastest permutation test and SRSWOR algorithm in SAS, but also the most robust. Given SAS's reputation for speed compared to other statistical computing platforms, OPDN is likely to be a serious contender for the fastest algorithm for these purposes among all major statistical computing packages.

That said, it should be noted that the algorithm runtimes presented herein, and in any empirical algorithm research for that matter, obviously very much depend on the hardware configurations on which the algorithms are run. Very fast I/O speeds on grid platforms, for example, may well allow the hashing algorithms to close the speed gap on OPDY, or Proc SurveySelect to close the speed gap on OPDN. However, it is also likely that such platforms also will have comparably increased memory resources, and thus, OPDY and OPDN may retain or even increase their speed lead: all else equal, as a rule, in-memory processing will always be faster than I/O processing, and OPDY and OPDN do not need to create and write to disk the bootstrap sampling dataset that the hashing algorithms "merge" with the original input dataset, nor the sampling dataset that Proc SurveySelect needs to generate, respectively. In fact, preliminary runs on such systems (with large amounts of memory) show dramatic (relative) speed gains for OPDY and OPDN beyond those presented herein when compared to most of the competitors used herein. The code provided in this paper allows for testing and comparisons on different platforms, and this author welcomes feedback from readers regarding the relative performance of the algorithms across an array of hardware configurations.

Acknowledgments

I sincerely thank Toyo Johnson and Nicole Ann Johnson Opdyke for their support and belief that SAS could produce better bootstraps, better permutation tests, and better sampling with and without replacement.

Appendix A

SAS v.9.2 code for the OPDN, OPDN-Alt, PROCSS, PROCMT, PROCNPAR, and Bebb-Sim algorithms, and the SAS code that generates the datasets used to test them in this paper, is presented below.

```
*************************************************************************
*************************************************************************
PROGRAM:  MFPTUS.SAS

DATE:  12/30/10

CODER:       J.D. Opdyke

PURPOSE:  Run and compare 6 different SAS Permutation Tests algorithms
          including OPDN, OPDN_alt, PROCSS, PROCMT, PROCNPAR, and BEBB_SIM.
             See Opdyke J.D., "Permutation Tests (and Sampling with Replacement) Orders of
             Magnitude Faster Using SAS," InterStat, January, 2011, for detailed
explanations
             of the different algorithms.

INPUTS:      Each macro is completely modular and accepts 6 macro parameter
          values (PROCMT accepts 7):
                   indata        = the input dataset (including libname)
                   outdata       = the input dataset (including libname)
                   byvars        = the "by variables" defining the strata (these are
                                   character variables)
                   samp2var      = the (character) variable defining the two samples in
each
```

```
                                    stratum: TEST ("T") and CONTROL ("C"), with those
values
                  permvar       = the variable to be tested with permutation tests
                  num_psmps     = number of Permutation Test samples
                  seed          = and optional random number seed (must be an integer or
                                    blank)
               left_or_right = "left" or "right" depending on whether the user wants
                                    lower or upper one-tailed p-values, respectively, in
                                    addition to the two-tailed p-value (only for PROCMT)

OUTPUTS:  A SAS dataset, named by the user via the macro variable outdata,
          which contains the following variables:
                  1) the "by variables" defining the strata
                  2) the name of the permuted variable
                  3) the size (# of records) of the permutation samples (this should
                     always be the smaller of the two samples (TEST v. CONTROL))
                  4) the number of permutation samples
                  5) the left, right, and two-tailed p-values corresponding to the
                     following hypotheses:
                        p_left  = Pr(x>X | Ho: Test>=Control)
                        p_right = Pr(x<X | Ho: Test<=Control)
                        p_both  = Pr(x=X | Ho: Test=Control)
                     where x is the sample permutation statistic from the Control
                     sample (or the additive inverse of the Test sample if the Test
                     sample is smaller, which is atypical) and X is the distribution of
                     permutation statistic values.

                     Following standard convention, the two-tailed p-value is
                     calculated based on the reflection method.
************************************************************************
************************************************************************
***;


options
label        symbolgen
fullstimer   yearcutoff=1950
nocenter     ls = 256     ps = 51
msymtabmax=max
mprint       mlogic
minoperator  mindelimiter=' '
cleanup
;


libname MFPTUS "c:\";

%MACRO makedata(strata_size=, testproportion=, numsegs=, numgeogs=);

*** Since generating all the datasets below once takes only a couple of minutes,
    time was not wasted trying to optimize runtimes for the creation of mere
       test data.
***;

%let numstrata = %eval(&numsegs.*&numgeogs.);

*** The variable "cntrl_test" identifies the test and control samples with
    values of "T" and "C" respectively.
***;
```

```
data MFPTUS.pricing_data_&numstrata.strata_&strata_size.(keep=geography segment cntrl_test
price sortedby=geography segment);
  format segment geography $8. cntrl_test $1.;
  array seg{3} $ _TEMPORARY_ ('segment1' 'segment2' 'segment3');
  array geog{4} $ _TEMPORARY_ ('geog1' 'geog2' 'geog3' 'geog4');
  strata_size = 1* &strata_size.;
  do x=1 to &numgeogs.;
     geography=geog{x};
     do j=1 to &numsegs.;
        segment=seg{j};
        if j=1 then do i=1 to strata_size;
              if mod(i,&testproportion.)=0 then do;
                 cntrl_test="T";
                 price=(rand('UNIFORM')+0.175/(&testproportion./(&testproportion./10)));
              end;
              else do;
                 cntrl_test="C";
                 price=rand('UNIFORM');
              end;
              output;
           end;
           else if j=2 then do i=1 to strata_size;
              if mod(i,&testproportion.)=0 then do;
                 cntrl_test="T";
                 price=rand('POISSON',1-0.75/(&testproportion./(&testproportion./10)));
              end;
              else do;
                 cntrl_test="C";
                 price=rand('POISSON',1.0);
              end;
              output;
           end;
           else if j=3 then do i=1 to strata_size;
*** Make Control smaller to test code in algorithms.;
              if mod(i,&testproportion.)=0 then cntrl_test="C";
              else cntrl_test="T";
              price=rand('NORMAL');
              output;
           end;
     end;
  end;
  run;
%MEND makedata;

%makedata(strata_size=10000, testproportion=10, numsegs=2, numgeogs=1);
%makedata(strata_size=10000, testproportion=10, numsegs=2, numgeogs=3);
%makedata(strata_size=10000, testproportion=10, numsegs=3, numgeogs=4);

%makedata(strata_size=100000, testproportion=100, numsegs=2, numgeogs=1);
%makedata(strata_size=100000, testproportion=100, numsegs=2, numgeogs=3);
%makedata(strata_size=100000, testproportion=100, numsegs=3, numgeogs=4);

%makedata(strata_size=1000000, testproportion=1000, numsegs=2, numgeogs=1);
%makedata(strata_size=1000000, testproportion=1000, numsegs=2, numgeogs=3);
%makedata(strata_size=1000000, testproportion=1000, numsegs=3, numgeogs=4);

%makedata(strata_size=10000000, testproportion=10000, numsegs=2, numgeogs=1);
%makedata(strata_size=10000000, testproportion=10000, numsegs=2, numgeogs=3);
%makedata(strata_size=10000000, testproportion=10000, numsegs=3, numgeogs=4);
```

```
%makedata(strata_size=7500000, testproportion=7500, numsegs=2, numgeogs=1);
%makedata(strata_size=7500000, testproportion=7500, numsegs=2, numgeogs=3);


%makedata(strata_size=25000000, testproportion=25000, numsegs=2, numgeogs=1);
%makedata(strata_size=50000000, testproportion=50000, numsegs=2, numgeogs=1);
%makedata(strata_size=100000000, testproportion=100000, numsegs=2, numgeogs=1);
```

***** OPDN ***;**
***** OPDN ***;**
***** OPDN ***;**

```
%MACRO OPDN(num_psmps=,
            indata=,
            outdata=,
            byvars=,
            samp2var=,
            permvar=,
            seed=
            );

*** The only assumption made within this macro is that the byvars are all
    character variables and the input dataset is sorted by the "by variables"
       that define the strata.  Also, the "TEST" and "CONTROL" samples are defined
    by a variable "cntrl_test" formatted as $1. containing "T" and "C"
       character strings, respectively, as values.
***;

*** Obtain the last byvar, count the byvars, and assign each byvar into numbered
    macro variables for easy access/processing.
***;

%let last_byvar = %scan(&byvars.,-1);
%let num_byvars = %sysfunc(countw(&byvars.));
%do i=1 %to &num_byvars.;
   %let byvar&i. = %scan(&byvars.,&i.);
%end;

*** If user does not pass a value to the optional macro variable "seed," use -1
    based on the time of day.
***;
%if %sysevalf(%superq(seed)=,boolean) %then %let seed=-1;

*** Obtain counts and cumulated counts for each strata.;

proc summary data=&indata. nway;
  class &byvars. &samp2var.;
  var &permvar.;
  output out=byvar_sum(keep = _FREQ_ &byvars. &samp2var. sumpvar)
                        sum = sumpvar
                       ;
  run;

*** Identify and keep the smaller of the two samples for more efficient
    sampling.  Below, invert the empirical permutation distribution if CONTROL
       sample is smaller that TEST sample (which is not typical).  That will
       remain consistent with output variables corresponding to:
```

```
            p_left  = Pr(x>X | Ho: Test>=Control)
            p_right = Pr(x<X | Ho: Test<=Control)
            p_both  = Pr(x=X | Ho: Test=Control)
       where x is the sample permutation statistic and X is the distribution of
       permutation statistic values.
***;

data byvar_sum_min(keep=tot_FREQ _FREQ_ &byvars. &samp2var. sumpvar sortedby=&byvars.);
  set byvar_sum;
  format tot_FREQ _FREQ_ 16.;
  by &byvars.;
  retain lag_FREQ lag_sum lag_samp2;
  if first.&last_byvar. then do;
        lag_FREQ = _FREQ_;
        lag_sum  = sumpvar;
        lag_samp2  = &samp2var.;
  end;
  else do;
     tot_FREQ = sum(lag_FREQ,_FREQ_);
     if _FREQ_<=lag_FREQ then output;
        else do;
           _FREQ_  = lag_FREQ;
           sumpvar = lag_sum;
           &samp2var. = lag_samp2;
           output;
        end;
  end;
  run;


*** Obtain number of strata, and use this number to count and separately permute
    each strata below.
***;

%let dsid = %sysfunc(open(byvar_sum_min));
%let n_byvals = %sysfunc(ifc(&dsid.
                                    ,%nrstr(%sysfunc(attrn(&dsid.,nobs))
                                            %let rc = %sysfunc(close(&dsid.));
                                               )
                                    ,%nrstr(0)
                                 )
                          );
*** In case number is large, avoid scientific notation.;
%let n_byvals = %sysfunc(putn(&n_byvals.,16.));


*** Cumulate counts of strata for efficient (re)use of _TEMPORARY_ array.;

data byvar_sum_min(drop=prev_freq);
  set byvar_sum_min;
  format cum_prev_freq _FREQ_ 16.;
  retain cum_prev_freq 0;
  prev_freq = lag(tot_FREQ);
  if _n_=1 then prev_freq = 0;
  cum_prev_freq = sum(cum_prev_freq, prev_freq);
  run;


*** For access in data step below (that uses data _null_ for memory
    conservation), put into macro strings a) smaller-of-two-sample counts,
       b) total counts for each strata, c) cumulated total counts, d) summed
```

```
       permutation variable, d) which of two samples is smaller, and e) byvar
       values.
***;

proc sql noprint;
  select _freq_ into :freqs separated by ' ' from byvar_sum_min;
  quit;
proc sql noprint;
  select tot_FREQ into :tot_FREQs separated by ' ' from byvar_sum_min;
  quit;
proc sql noprint;
  select cum_prev_freq into :cum_prev_freqs separated by ' ' from byvar_sum_min;
  quit;
proc sql noprint;
  select sumpvar into :sumpvar separated by ' ' from byvar_sum_min;
  quit;
proc sql noprint;
  select &samp2var. into :samp2var separated by ' ' from byvar_sum_min;
  quit;

%do i=1 %to &num_byvars.;
   proc sql noprint;
     select &&byvar&i. into :byvals&i. separated by ' ' from byvar_sum_min;
     quit;
%end;


*** Get size of largest stratum for efficient (re)use of _TEMPORARY_ array.;

proc sql noprint;
  select max(tot_FREQ) into :max_tot_freq separated by ' ' from byvar_sum_min;
  quit;

*** In case number is large, avoid scientific notation.;
%let max_tot_freq = %sysfunc(putn(&max_tot_freq.,16.));


*** Save each stratums results in cumulated macro variables instead of
    outputting to a dataset on the data step to lessen intermediate memory
       requirements.
*** ;

*** Initialize macro variables used below.;
%let p_left =;
%let p_right =;
%let p_both =;
%let samp_small_size =;

data _null_;
  set &indata.;
  by &byvars.;
  format which_sample $1.;
*** To view permutation distributions for each strata in .log file.,
    comment out first line below this comment, uncomment the two lines below
       it, and uncomment "put _ALL_" 37 lines below it.  Note that this will slow
       program execution and often create large .log files.
***;

  array temp{&max_tot_freq.} _TEMPORARY_;
%if &sysver >= 9.2 %then %do;
  array psums{&num_psmps.} _TEMPORARY_;
```

```
  retain byval_counter 0 cum_prev_freq 0;
%end;
%if &sysver < 9.2 %then %do;
  array psums{&num_psmps.} psmp1-psmp&num_psmps.;
  retain byval_counter 0 cum_prev_freq 0 psmp1-psmp&num_psmps.;
%end;

  temp[_n_-cum_prev_freq]=&permvar.;

  if last.&last_byvar. then do;
     byval_counter+1;
     num_psmps = &num_psmps.*1;
     psmp_size = 1 * scan("&freqs.", byval_counter,' ');
        which_sample = COMPRESS(UPCASE(scan("&samp2var.", byval_counter,' ')),' ');
     tot_FREQ_hold = 1 * scan("&tot_FREQs.", byval_counter,' ');
        seed = 1*&seed.;

     do m=1 to num_psmps;
        x=0;
        tot_FREQ = tot_FREQ_hold;
        do n=1 to psmp_size;
              cell = floor(ranuni(seed)*tot_FREQ) + 1;
              x = temp[cell] + x;
              hold = temp[cell];
              temp[cell]=temp[tot_FREQ];
              temp[tot_FREQ] = hold;
              tot_FREQ+(-1);
           end;
           psums[m] = x;
     end;
     psum = 1*scan("&sumpvar.", byval_counter,' ');
     p_right = 0;
        p_left  = 0;
        p_both  = 0;
        call sortn(of psums[*]);
     pmed = median(of psums[*]);
     pmean = mean(of psums[*]);

* put _ALL_;

*** Efficiently handle extreme test sample values.;

     IF psum<psums[1] THEN DO;
           p_left=0;
           p_right=num_psmps;
           p_both=0;
        END;
        ELSE IF psum>psums[num_psmps] THEN DO;
           p_left=num_psmps;
           p_right=0;
           p_both=0;
        END;
        ELSE DO;

*** For non-extreme cases, start with shorter tail for less looping.;

     if pmed>=psum then do;
           do z=1 to num_psmps;
              if psum>=psums[z] then p_left+1;
              else do;
                 lastbinnum = z-1;
```

```
                distance_left = pmean - psums[z-1];
                leave;
             end;
          end;

*** Avoid loop for other (larger) p-value.
    If psum equals last bin, p_right = 1 - p_left + lastbinsize.
    Otherwise, p_right = 1 - p_left.
***;
          if psum = psums[lastbinnum] then do;
             lastbinsize=1;
             do k=lastbinnum to 1 by -1;
                if psums[k]=psums[k-1] then lastbinsize+1;
                leave;
             end;
             p_right = num_psmps - p_left + lastbinsize;
          end;
          else p_right = num_psmps - p_left;
        end;

        else do;
           do z=num_psmps to 1 by -1;
              if psum<=psums[z] then p_right+1;
              else do;
                 lastbinnum = z+1;
                 distance_right = psums[z+1] - pmean;
                 leave;
              end;
           end;

*** Avoid loop for other (larger) p-value.
    If psum equals last bin, p_left = 1 - p_right + lastbinsize.
    Otherwise, p_left = 1 - p_right.
***;
           if psum = psums[lastbinnum] then do;
              lastbinsize=1;
              do k=lastbinnum to num_psmps;
                 if psums[k]=psums[k+1] then lastbinsize+1;
                 else leave;
              end;
              p_left = num_psmps - p_right + lastbinsize;
           end;
           else p_left = num_psmps - p_right;
         end;

*** Base 2-sided p-value on distance from mean of last (i.e. least extreme) bin
    of smaller p-value.  This is common practice.
***;

         if p_left<p_right then do;
            p_both = p_left;
         do z=num_psmps to 1 by -1;
               if (psums[z] - pmean) >= distance_left then p_both+1;
               else leave;
            end;
         end;
         else if p_left>p_right then do;
            p_both = p_right;
         do z=1 to num_psmps;
               if (pmean - psums[z]) >= distance_right then p_both+1;
               else leave;
```

```
            end;
         end;
         else p_both=num_psmps;

*** Account for possibility, due to psum=a particular bin value, that
    p_both>num_psmps.
***;
         p_both = min(p_both,num_psmps);

        END;

        p_left  = p_left  / num_psmps;
        p_right = p_right / num_psmps;
        p_both  = p_both  / num_psmps;

*** If CONTROL sample is smaller than TEST (which is atypical), reverse
*** p-values, as empirical distribution is mirror of itself.;

     if "C"=COMPRESS(UPCASE(scan("&samp2var.", byval_counter,' ')),' ') then do;
           hold = p_left;
           p_left = p_right;
           p_right = hold;
        end;

*** Cumulate key macro variables to save results.;

     call symput('p_left',symget('p_left')||" "||compress(p_left));
     call symput('p_right',symget('p_right')||" "||compress(p_right));
     call symput('p_both',symget('p_both')||" "||compress(p_both));
        cum_prev_freq = 1*scan("&cum_prev_freqs.",byval_counter+1,' ');
  end;
  run;

*** Obtain and assign the format of each byvar, all of which are assumed to be
    character variables.
***;

data lens(keep=lens);
  set &indata.(keep=&byvars. firstobs=1 obs=1);
  do i=1 to &num_byvars.;
     lens = vlengthx(scan("&byvars.",i));
        output;
  end;
  run;
proc sql noprint;
  select lens into :alllens separated by ' ' from lens;
  quit;
%macro assign_formats;
   %do i=1 %to &num_byvars.;
      &&byvar&i. $%scan(&alllens.,&i.).
      %end;
%mend assign_formats;


*** Assign each byvar value for each stratum.;

%macro assign_byvar_vals(which_strata=);
   %do j=1 %to &num_byvars.;
         &&byvar&j. = scan("&&byvals&j.",&which_strata.,' ');
   %end;
%mend assign_byvar_vals;
```

```
*** Unwind and assign all the cumulated macro variables.;

data &outdata.(sortedby=&byvars. drop=n_byvals i);
  n_byvals = 1*&n_byvals.;
  format %assign_formats;
  do i=1 to n_byvals;
     length permvar $32;
     permvar = "&permvar.";
     n_psamp = 1 * scan("&freqs.", i,' ');
     num_psmps = &num_psmps.;
     p_left = 1*scan("&p_left.",i,' ');
     p_right = 1*scan("&p_right.",i,' ');
     p_both = 1*scan("&p_both.",i,' ');
        %assign_byvar_vals(which_strata = i)
     label permvar   = "Permuted Variable"
           n_psamp   = "Size of Permutation Samples"
           num_psmps = "# of Permutation Samples"
           p_left    = "Left p-value"
           p_right   = "Right p-value"
           p_both    = "Two-Tailed p-value"
           ;
     output;
  end;
  run;


*** Optional.;
* proc datasets lib=work memtype=data kill nodetails;
*   run;

%MEND OPDN;

%OPDN(num_psmps = 1000,
      indata    = MFPTUS.pricing_data_2strata_100000,
      outdata   = MFPTUS.OPDN_100000_2strata,
      byvars    = geography segment,
      samp2var  = cntrl_test,
      permvar   = price,
      seed      =
     );
```

***** OPDN_alt ***;**
***** OPDN_alt ***;**
***** OPDN_alt ***;**

```
%MACRO OPDN_alt(num_psmps=,
                indata=,
                   outdata=,
                   byvars=,
                   samp2var=,
                   permvar=,
                   seed=
                   );

*** If user does not pass a value to the optional macro variable "seed," use -1
    based on the time of day.
```

```
***;
%if %sysevalf(%superq(seed)=,boolean) %then %let seed=-1;

*** To minimize intermediate memory requirements, initialize output data set
    with missing variable values.
***;

data &outdata.(sortedby=&byvars.);
   stop;
   set &indata(keep=&byvars.);
   length permvar $32 n_psamp num_psmps p_left p_right p_both
          distance_right distance_left lastbinnum lastbinsize
          pmed pmean tot_FREQ_incr x 8;
   call missing(of _all_);
   run;

*** Obtain counts and cumulated counts for each strata.;

proc summary data=&indata. nway;
  class &byvars. &samp2var.;
  var &permvar.;
  output out=byvar_sum(keep = _FREQ_ &byvars. &samp2var. sumpvar)
                       sum = sumpvar;
  run;

*** Identify and keep the smaller of the two samples for more efficient
    sampling.  Below, invert the empirical permutation distribution if CONTROL
       sample is smaller that TEST sample (which is not typical).  That will
       remain consistent with output variables corresponding to:
            p_left  = Pr(x>X | Ho: Test>=Control)
            p_right = Pr(x<X | Ho: Test<=Control)
            p_both  = Pr(x=X | Ho: Test=Control)
       where x is the sample permutation statistic and X is the distribution of
       permutation statistic values.
***;

%let last_byvar = %scan(&byvars.,-1);

data byvar_sum_min(keep=tot_FREQ _FREQ_ &byvars. &samp2var. sumpvar sortedby=&byvars.);
  set byvar_sum;
  format tot_FREQ _FREQ_ 16.;
  by &byvars.;
  retain lag_FREQ lag_sum lag_samp2;
  if first.&last_byvar. then do;
        lag_FREQ = _FREQ_;
        lag_sum  = sumpvar;
        lag_samp2  = &samp2var.;
  end;
  else do;
     tot_FREQ = sum(lag_FREQ,_FREQ_);
     if _FREQ_<=lag_FREQ then output;
        else do;
           _FREQ_  = lag_FREQ;
           sumpvar = lag_sum;
           &samp2var. = lag_samp2;
           output;
        end;
  end;
  run;
```

```
*** Get size of largest stratum for efficient (re)use of _TEMPORARY_ array.;

proc sql noprint;
   select max(tot_FREQ) into :max_tot_freq from byvar_sum_min;
   quit;

*** In case number is large, avoid scientific notation.;
%let max_tot_freq = %sysfunc(putn(&max_tot_freq.,16.));

data &outdata.;
   if 0 then modify &outdata.;

*** To view permutation distributions for each strata in .log file.,
    comment out first line below this comment, uncomment the line below it, and
    uncomment "put _ALL_" 35 lines below it.  Note that this will slow program
       execution and often create large .log files.
***;
  array temp{&max_tot_freq.} _TEMPORARY_;
%if &sysver >= 9.2 %then %do;
  array psums{&num_psmps.} _TEMPORARY_;
  retain num_psmps &num_psmps.;
%end;
%if &sysver < 9.2 %then %do;
  array psums{&num_psmps.} psmp1-psmp&num_psmps.;
  retain num_psmps &num_psmps. psmp1-psmp&num_psmps.;
%end;

   do _n_ = 1 by 1 until(last.&last_byvar.);
      merge &indata.(keep=&byvars. &permvar. &samp2var.)
               byvar_sum_min;
      by &byvars.;
      temp[_n_]=&permvar.;
   end;
   seed = 1*&seed.;
   do m=1 to num_psmps;
      x=0;
      tot_FREQ_incr = tot_FREQ;
      do n=1 to _FREQ_;
            cell = floor(ranuni(seed)*tot_FREQ_incr) + 1;
            x = temp[cell] + x;
            hold = temp[cell];
            temp[cell]=temp[tot_FREQ_incr];
            temp[tot_FREQ_incr] = hold;
            tot_FREQ_incr+(-1);
         end;
      psums[m] = x;
   end;

   n_psamp = _FREQ_;

   p_right = 0;
   p_left  = 0;
      p_both  = 0;
      call sortn(of psums[*]);
   pmed = median(of psums[*]);
   pmean = mean(of psums[*]);

* put _ALL_;

*** Efficiently handle extreme test sample values.;
```

```
   IF sumpvar<psums[1] THEN DO;
      p_left=0;
      p_right=num_psmps;
      p_both=0;
   END;
   ELSE IF sumpvar>psums[num_psmps] THEN DO;
      p_left=num_psmps;
      p_right=0;
      p_both=0;
   END;
   ELSE DO;

*** For non-extreme cases, start with shorter tail for less looping.;

   if pmed>=sumpvar then do;
      do z=1 to num_psmps;
         if sumpvar>=psums[z] then p_left+1;
         else do;
            lastbinnum = z-1;
            distance_left = pmean - psums[z-1];
            leave;
         end;
      end;

*** Avoid loop for other (larger) p-value.
    If sumpvar equals last bin, p_right = 1 - p_left + lastbinsize.
    Otherwise, p_right = 1 - p_left.
***;
      if sumpvar = psums[lastbinnum] then do;
         lastbinsize=1;
         do k=lastbinnum to 1 by -1;
            if psums[k]=psums[k-1] then lastbinsize+1;
            leave;
         end;
         p_right = num_psmps - p_left + lastbinsize;
      end;
      else p_right = num_psmps - p_left;
   end;
   else do;
      do z=num_psmps to 1 by -1;
         if sumpvar<=psums[z] then p_right+1;
         else do;
            lastbinnum = z+1;
            distance_right = psums[z+1] - pmean;
            leave;
         end;
      end;

*** Avoid loop for other (larger) p-value.
    If psum equals last bin, p_left = 1 - p_right + lastbinsize.
    Otherwise, p_left = 1 - p_right.
***;
      if sumpvar = psums[lastbinnum] then do;
         lastbinsize=1;
         do k=lastbinnum to num_psmps;
            if psums[k]=psums[k+1] then lastbinsize+1;
            else leave;
         end;
         p_left = num_psmps - p_right + lastbinsize;
      end;
      else p_left = num_psmps - p_right;
```

```
   end;

*** Base 2-sided p-value on distance from mean of last (i.e. least extreme) bin
    of smaller p-value.  This is common practice.
***;

   if p_left<p_right then do;
      p_both = p_left;
      do z=num_psmps to 1 by -1;
         if (psums[z] - pmean) >= distance_left then p_both+1;
         else leave;
      end;
   end;
   else if p_left>p_right then do;
      p_both = p_right;
      do z=1 to num_psmps;
         if (pmean - psums[z]) >= distance_right then p_both+1;
         else leave;
      end;
   end;
   else p_both=num_psmps;

*** Account for possibility, due to psum=a particular bin value, that p_both>num_psmps.
***;
   p_both = min(p_both,num_psmps);

   END;

*** If CONTROL sample is smaller than TEST (which is atypical), reverse
*** p-values, as empirical distribution is mirror of itself.;

   if &samp2var.="C" then do;
      hold = p_left;
         p_left = p_right;
         p_right = hold;
      end;

   p_left  = p_left  / num_psmps;
   p_right = p_right / num_psmps;
   p_both  = p_both  / num_psmps;

   length permvar $32;
   retain permvar "&permvar";
   output;
   run;

data &outdata.;
   set &outdata.(keep=&byvars. permvar num_psmps n_psamp p_left p_right p_both);
   label permvar   = "Permuted Variable"
         n_psamp   = "Size of Permutation Samples"
         num_psmps = "# of Permutation Samples"
         p_left    = "Left p-value"
         p_right   = "Right p-value"
         p_both    = "Two-Tailed p-value"
         ;
   run;


*** Optional.;
* proc datasets lib=work memtype=data kill nodetails;
*   run;
```

```
%MEND OPDN_alt;


%OPDN_alt(num_psmps = 1000,
          indata    = MFPTUS.pricing_data_2strata_100000,
          outdata   = MFPTUS.OPDN_alt_100000_2strata,
          byvars    = geography segment,
          samp2var  = cntrl_test,
          permvar   = price,
          seed      =
         );
```

***** PROC_SS ***;**
***** PROC_SS ***;**
***** PROC_SS ***;**

```
%MACRO PROCSS(num_psmps=,
              indata=,
              outdata=,
              byvars=,
              samp2var=,
              permvar=,
              seed=
             );


*** If user does not pass a value to the optional macro variable "seed," use -1
    based on the time of day.
***;
%if %sysevalf(%superq(seed)=,boolean) %then %let seed=-1;

*** Obtain counts and cumulated counts for each strata.;

proc summary data=&indata. nway;
  class &byvars. &samp2var.;
  var &permvar.;
  output out=byvar_sum(keep = _FREQ_ &byvars. &samp2var. sumpvar)
                        sum = sumpvar
                       ;
  run;

*** Identify and keep the smaller of the two samples for more efficient
    sampling.  Below, invert the empirical permutation distribution if CONTROL
       sample is smaller that TEST sample (which is not typical).  That will
       remain consistent with output variables corresponding to:
            p_left  = Pr(x>X | Ho: Test>=Control)
            p_right = Pr(x<X | Ho: Test<=Control)
            p_both  = Pr(x=X | Ho: Test=Control)
       where x is the sample permutation statistic and X is the distribution of
       permutation statistic values.
***;

%let last_byvar = %scan(&byvars.,-1);
data byvar_sum(keep=&byvars. _NSIZE_ sumpvar &samp2var. sortedby=&byvars.);
  set byvar_sum(rename=(_FREQ_=_NSIZE_));
  by &byvars.;
```

```
  retain lag_NSIZE lag_sum lag_samp2;
  if first.&last_byvar. then do;
     lag_NSIZE = _NSIZE_;
        lag_sum  = sumpvar;
        lag_samp2  = &samp2var.;
  end;
  else do;
     if _NSIZE_<=lag_NSIZE then output;
        else do;
              _NSIZE_  = lag_NSIZE;
              sumpvar = lag_sum;
              &samp2var. = lag_samp2;
              output;
        end;
  end;
  run;

*** From SAS Online Documentation:
    For simple random sampling without replacement, if there is enough memory
    for it PROC SURVEYSELECT uses Floyds ordered hash table algorithm (see
       Bentley and Floyd (1987) and Bentley and Knuth (1986) for details).  If
       there is not enough memory available for Floyds algorithm, PROC
       SURVEYSELECT switches to the sequential algorithm of Fan, Muller, and
       Rezucha (1962), which requires less memory but might require more time to
       select the sample.
***;

proc surveyselect data     = &indata.(drop=&samp2var.)
                  method   = srs
                     sampsize = byvar_sum(keep=&byvars. _NSIZE_)
                     rep      = &num_psmps.
                     seed     = &seed.
                     out      = PSS_perm_Samps(drop=SamplingWeight SelectionProb)
                     noprint;
  strata &byvars.;
  run;

proc summary data=PSS_perm_Samps nway;
  class &byvars. replicate;
  var &permvar.;
output out=PSS_perm_sums(sortedby=&byvars. replicate keep=&byvars. replicate &permvar.)
sum=;
  run;

proc transpose data=PSS_perm_sums out=PSS_perm_sums_t(rename=(_NAME_=permvar))
prefix=psmp;
  var &permvar.;
  by &byvars.;
  id replicate;
  run;

data &outdata.(keep=&byvars. permvar n_psamp num_psmps p_left p_right p_both)
     error
        ;
  merge PSS_perm_sums_t(in=insamps)
        byvar_sum(in=insummary)
           ;
  by &byvars.;
  if insamps & insummary then do;
     array psums[&num_psmps.] psmp1-psmp&num_psmps.;
     n_psamp = _NSIZE_;
```

```
     num_psmps = 1*&num_psmps.;
     p_left  = 0;
     p_right = 0;
     p_both  = 0;
     call sortn(of psums[*]);
     pmed = median(of psums[*]);
     pmean = mean(of psums[*]);

*** Efficiently handle extreme test sample values.;

     IF sumpvar<psums[1] THEN DO;
        p_left=0;
        p_right=num_psmps;
        p_both=0;
     END;
     ELSE IF sumpvar>psums[num_psmps] THEN DO;
        p_left=num_psmps;
        p_right=0;
        p_both=0;
     END;
     ELSE DO;

*** For non-extreme cases, start with shorter tail for less looping.;

        if pmed>=sumpvar then do;
           do z=1 to num_psmps;
              if sumpvar>=psums[z] then p_left+1;
              else do;
                 lastbinnum = z-1;
                 distance_left = pmean - psums[z-1];
                 leave;
              end;
           end;

*** Avoid loop for other (larger) p-value.
    If sumpvar equals last bin, p_right = 1 - p_left + lastbinsize.
    Otherwise, p_right = 1 - p_left.
***;
           if sumpvar = psums[lastbinnum] then do;
              lastbinsize=1;
              do k=lastbinnum to 1 by -1;
                 if psums[k]=psums[k-1] then lastbinsize+1;
                 leave;
              end;
              p_right = num_psmps - p_left + lastbinsize;
           end;
           else p_right = num_psmps - p_left;
        end;

        else do;
           do z=num_psmps to 1 by -1;
              if sumpvar<=psums[z] then p_right+1;
              else do;
                 lastbinnum = z+1;
                 distance_right = psums[z+1] - pmean;
                 leave;
              end;
           end;

*** Avoid loop for other (larger) p-value.
    If psum equals last bin, p_left = 1 - p_right + lastbinsize.
```

```
    Otherwise, p_left = 1 - p_right.
***;
           if sumpvar = psums[lastbinnum] then do;
              lastbinsize=1;
              do k=lastbinnum to num_psmps;
                 if psums[k]=psums[k+1] then lastbinsize+1;
                 else leave;
              end;
              p_left = num_psmps - p_right + lastbinsize;
           end;
           else p_left = num_psmps - p_right;
        end;

*** Base 2-sided p-value on distance from mean of last (i.e. least extreme) bin
    of smaller p-value.  This is common practice.
***;

        if p_left<p_right then do;
           p_both = p_left;
           do z=num_psmps to 1 by -1;
              if (psums[z] - pmean) >= distance_left then p_both+1;
              else leave;
           end;
        end;
        else if p_left>p_right then do;
           p_both = p_right;
           do z=1 to num_psmps;
              if (pmean - psums[z]) >= distance_right then p_both+1;
              else leave;
           end;
        end;
        else p_both=num_psmps;

*** Account for possibility, due to psum=a particular bin value, that
    p_both>num_psmps.
***;
        p_both = min(p_both,num_psmps);

     END;

     p_left  = p_left  / num_psmps;
     p_right = p_right / num_psmps;
     p_both  = p_both  / num_psmps;

*** If CONTROL sample is smaller than TEST (which is atypical), reverse
*** p-values, as empirical distribution is mirror of itself.;

     if &samp2var.="C" then do;
           hold = p_left;
           p_left = p_right;
           p_right = hold;
        end;

     label permvar   = "Permuted Variable"
           n_psamp   = "Size of Permutation Samples"
           num_psmps = "# of Permutation Samples"
           p_left    = "Left p-value"
           p_right   = "Right p-value"
           p_both    = "Two-Tailed p-value"
           ;
     output &outdata.;
```

```
  end;
  else output error;
  run;

*** Optional ***;
* proc datasets lib=work memtype=data kill nodetails;
*   run;

%MEND PROCSS;

%PROCSS(num_psmps = 1000,
        indata    = MFPTUS.pricing_data_2strata_100000,
        outdata   = MFPTUS.PROCSS_100000_2strata,
        byvars    = geography segment,
        samp2var  = cntrl_test,
        permvar   = price,
        seed      =
        );
```

***** PROC_MT ***;**
***** PROC_MT ***;**
***** PROC_MT ***;**

```
%MACRO PROCMT(num_psmps=,
              indata=,
              outdata=,
              byvars=,
              samp2var=,
              permvar=,
              seed=,
                 left_or_right=
              );


*** If user does not pass a value to the optional macro variable "seed,"
    generate a random seed and use it for all three proc multtests below
       (although SAS OnlineDoc says seed=-1 should work, it does not).
***;
%if %sysevalf(%superq(seed)=,boolean)
%then %let seed=%sysfunc(ceil(%sysevalf(1000000000*%sysfunc(ranuni(-1)))));

*** Un/comment test statements below to perform permutation tests based
    on different assumptions about the variance structure of the samples.;
***;

proc multtest    data = &indata.
              nsample = &num_psmps.
                    seed = &seed.
                     out = mt_output_results_2t(keep=&byvars. perm_p
rename=(perm_p=xp_both))
                    permutation
                      noprint;
  by &byvars.;
  class &samp2var.;
*  test mean (&permvar. / DDFM=SATTERTHWAITE);
  test mean (&permvar.);
  run;
```

```
*** To make runtime results comparable to PROC NPAR1WAY, which provides only
    two-tailed p-value and the smaller of the right or left p-values, run
       two of the three PROC MULTTESTs and calculate the second tail as one
       minus the given tail, which will usually be very close to the actual
       value unless the data is highly discretized.
***;

proc multtest    data = &indata.
              nsample = &num_psmps.
                   seed = &seed.
%if %UPCASE(%sysfunc(compress(&left_or_right.)))=RIGHT %then %do;
                    out = mt_output_results_up(keep=&byvars. perm_p
rename=(perm_p=xp_right))
%end;
%if %UPCASE(%sysfunc(compress(&left_or_right.)))=LEFT %then %do;
               out = mt_output_results_low(keep=&byvars. perm_p rename=(perm_p=xp_left))
%end;
                    permutation
                      noprint;
  by &byvars.;
  class &samp2var.;
%if %UPCASE(%sysfunc(compress(&left_or_right.)))=RIGHT %then %do;
*  test mean (&permvar. / upper DDFM=SATTERTHWAITE);
  test mean (&permvar. / upper);
%end;
%if %UPCASE(%sysfunc(compress(&left_or_right.)))=LEFT %then %do;
*  test mean (&permvar. / lower DDFM=SATTERTHWAITE);
  test mean (&permvar. / lower);
%end;
  run;

proc summary data=&indata. nway;
  class &byvars. &samp2var.;
  var &permvar.;
  output out=byvar_frq(keep = _FREQ_ &byvars.)
                          n = toss
                          ;
  run;

%let last_byvar = %scan(&byvars.,-1);
data byvar_frq(keep=&byvars. xn_psamp sortedby=&byvars.);
  set byvar_frq(rename=(_FREQ_=xn_psamp));
  by &byvars.;
  retain lag_FREQ;
  if first.&last_byvar. then lag_FREQ = xn_psamp;
  else do;
     if xn_psamp<=lag_FREQ then output;
        else do;
           xn_psamp  = lag_FREQ;
           output;
        end;
  end;
  run;


data &outdata.(drop=xn_psamp xp_left xp_both)
     error
        ;
%if %UPCASE(%sysfunc(compress(&left_or_right.)))=LEFT %then %do;
```

```
  merge mt_output_results_low(in=inlow)
              mt_output_results_2t(in=in2t)
        byvar_frq(in=infrq)
        ;
  if in2t & inlow & infrq then do;
     format permvar $32.;
     permvar = "&permvar.";
        n_psamp = xn_psamp;
     num_psmps = 1*&num_psmps.;
        p_left = xp_left;
        p_right = .;
%end;
%if %UPCASE(%sysfunc(compress(&left_or_right.)))=RIGHT %then %do;
  merge mt_output_results_up(in=inup)
              mt_output_results_2t(in=in2t)
        byvar_frq(in=infrq)
        ;
  if in2t & inup & infrq then do;
     format permvar $32.;
     permvar = "&permvar.";
        n_psamp = xn_psamp;
     num_psmps = 1*&num_psmps.;
        p_right = xp_right;
        p_left = .;
%end;
        p_both = xp_both;
     label permvar   = "Permuted Variable"
           n_psamp   = "Size of Permutation Samples"
           num_psmps = "# of Permutation Samples"
           p_left    = "Left p-value"
           p_right   = "Right p-value"
           p_both    = "Two-Tailed p-value"
           ;
     output &outdata.;
  end;
  else output error;
  run;


*** Optional ***;
* proc datasets lib=work memtype=data kill nodetails;
*   run;

%MEND PROCMT;

%PROCMT(num_psmps     = 1000,
        indata        = MFPTUS.pricing_data_2strata_100000,
        outdata       = MFPTUS.PROCMT_100000_2strata,
        byvars        = geography segment,
        samp2var      = cntrl_test,
        permvar       = price,
        seed          = ,
           left_or_right = left
        );
```

***** PROCNPAR ***;**

***** PROCNPAR ***;**

***** PROCNPAR ***;**

```
%MACRO PROCNPAR(num_psmps=,
                indata=,
                outdata=,
                byvars=,
                samp2var=,
                permvar=,
                seed=
                );

*** If user does not pass a value to the optional macro variable "seed," use -1
    based on the time of day.
***;
%if %sysevalf(%superq(seed)=,boolean) %then %let seed=-1;

ods listing close;
proc npar1way    data = &indata.
               scores = data
                              ;
  var &permvar.;
  by &byvars.;
  class &samp2var.;
  exact scores=data / n=&num_psmps. seed=&seed.;
  ods output DataScoresMC = hold(keep = &byvars. Name1 Label1 nValue1
                                 where = (Label1 = "Estimate")
                                 );
  run;
ods listing;

proc transpose data = hold(drop=Label1)
                out = &outdata.(drop = _NAME_);
  by &byvars.;
  id Name1;
  var nValue1;
  run;

proc summary data=&indata. nway;
  class &byvars. &samp2var.;
  var &permvar.;
  output out=byvar_frq(keep = _FREQ_ &byvars. &samp2var.)
                          n = toss
                          ;
  run;

%let last_byvar = %scan(&byvars.,-1);
data byvar_frq(keep=&byvars. permvar n_psamp num_psmps &samp2var. sortedby=&byvars.);
  set byvar_frq;
  by &byvars.;
  format permvar $32.;
  retain permvar "&permvar." lag_FREQ lag_samp2 ;
  n_psamp = _FREQ_;
  num_psmps = 1*&num_psmps.;
  if first.&last_byvar. then do;
     lag_FREQ = n_psamp;
        lag_samp2  = &samp2var.;
  end;
  else do;
     if n_psamp<=lag_FREQ then output;
        else do;
```

```
        n_psamp  = lag_FREQ;
           &samp2var. = lag_samp2;
           output;
        end;
  end;
  run;

data &outdata.(drop=hold mcpl_data mcpr_data mcp2_data &samp2var.)
     error
        ;
  merge byvar_frq(in=infrq)
           &outdata.(in=inresults)
        ;
  by &byvars.;
  if inresults & infrq then do;
  p_left  = mcpl_data;
  p_right = mcpr_data;
  p_both  = mcp2_data;
  label permvar   = "Permuted Variable"
        n_psamp   = "Size of Permutation Samples"
        num_psmps = "# of Permutation Samples"
        p_left    = "Left p-value"
        p_right   = "Right p-value"
        p_both    = "Two-Tailed p-value"
        ;

*** If CONTROL sample is smaller than TEST (which is atypical), reverse
*** p-values, as empirical distribution is mirror of itself.;

   if &samp2var.="C" then do;
      hold = p_left;
         p_left = p_right;
         p_right = hold;
   end;
      output &outdata.;
      end;
      else output error;
  run;

%MEND PROCNPAR;

%PROCNPAR(num_psmps = 1000,
          indata    = MFPTUS.pricing_data_2strata_100000,
          outdata   = MFPTUS.PROCNPAR_100000_2strata,
          byvars    = geography segment,
          samp2var  = cntrl_test,
          permvar   = price,
          seed      =
          );
```

***** BEBB_SIM ***;**
***** BEBB_SIM ***;**
***** BEBB_SIM ***;**

```
%MACRO BEBB_SIM(num_psmps=,
                indata=,
                outdata=,
```

```
                byvars=,
                samp2var=,
                permvar=,
                seed=
                );

*** If user does not pass a value to the optional macro variable "seed," use -1
    based on the time of day.
***;
%if %sysevalf(%superq(seed)=,boolean) %then %let seed=-1;


*** Obtain counts for each strata.;

proc summary data=&indata. nway;
  class &byvars. &samp2var.;
  var &permvar.;
  output out=byvar_sum(keep = _FREQ_ &byvars. &samp2var. sumpvar)
                        sum = sumpvar
                       ;
  run;

%let last_byvar = %scan(&byvars.,-1);
data byvar_sum_min(keep=tot_FREQ _FREQ_ &byvars. &samp2var. sumpvar sortedby=&byvars.);
  set byvar_sum;
  by &byvars.;
  retain lag_FREQ lag_sum lag_samp2;
  if first.&last_byvar. then do;
        lag_FREQ = _FREQ_;
        lag_sum  = sumpvar;
        lag_samp2  = &samp2var.;
  end;
  else do;
     tot_FREQ = sum(lag_FREQ,_FREQ_);
     if _FREQ_<=lag_FREQ then output;
        else do;
              _FREQ_  = lag_FREQ;
              sumpvar = lag_sum;
              &samp2var. = lag_samp2;
              output;
        end;
  end;
  run;

*** Slightly faster to use macro variable value strings and scan() than to merge
    _FREQ_ etc. onto the "indata" dataset, especially for large "indata"
       datasets.
*** ;
proc sql noprint;
  select _freq_ into :freqs separated by ' ' from byvar_sum_min;
  quit;
proc sql noprint;
  select tot_FREQ into :tot_FREQs separated by ' ' from byvar_sum_min;
  quit;
proc sql noprint;
  select sumpvar into :sumpvar separated by ' ' from byvar_sum_min;
  quit;
proc sql noprint;
  select &samp2var. into :samp2var separated by ' ' from byvar_sum_min;
  quit;
```

```
*** simultaneously create all permstrap samples and summarize results of each
    as the end of stratum is reached
***;

%let last_byvar = %scan(&byvars.,-1);

data &outdata.(keep=&byvars. permvar n_psamp num_psmps p_left p_right p_both);
  set &indata.(keep=&byvars. &permvar.) end=lastrec;
  by &byvars.;
  retain permvar "&permvar" n_psamp 0 num_psmps &num_psmps.;

  array smalln_counter{&num_psmps.} _TEMPORARY_;
  array psums{&num_psmps.} _TEMPORARY_;

  if first.&last_byvar. then do;

        byval_counter+1;

     _freq_ = 1*scan("&freqs.",byval_counter,' ');
     n_psamp = _FREQ_;
     tot_FREQ = 1*scan("&tot_FREQs.",byval_counter,' ');

        bigN_counter = tot_FREQ+1;
        do i=1 to num_psmps;
           smalln_counter[i] = _FREQ_;
           psums[i] = 0;
        end;
  end;
  bigN_counter+(-1);

  min_mac_res_inloop = &permvar.;

  seed=1*&seed.;
  if last.&last_byvar.~=1 then do i=1 to num_psmps;
     if ranuni(seed) <= smalln_counter[i]/bigN_counter then do;
        psums[i] = min_mac_res_inloop + psums[i];
           smalln_counter[i]+(-1);
        end;
  end;
  else do i=1 to num_psmps;
     if smalln_counter[i]>0 then psums[i] = min_mac_res_inloop + psums[i];
  end;
  if last.&last_byvar. then DO;


*** If CONTROL sample is smaller than TEST (which is atypical), reverse
*** order of empirical distribution.;

     sumpvar = 1*scan("&sumpvar.",byval_counter,' ');

     p_left  = 0;
     p_right = 0;
     p_both  = 0;
     call sortn(of psums[*]);
     pmed = median(of psums[*]);
     pmean = mean(of psums[*]);

*** Efficiently handle extreme test sample values.;

     IF sumpvar<psums[1] THEN DO;
```

```
        p_left=0;
        p_right=num_psmps;
        p_both=0;
     END;
     ELSE IF sumpvar>psums[num_psmps] THEN DO;
        p_left=num_psmps;
        p_right=0;
        p_both=0;
     END;
     ELSE DO;

*** For non-extreme cases, start with shorter tail for less looping.;

     if pmed>=sumpvar then do;
        do z=1 to num_psmps;
           if sumpvar>=psums[z] then p_left+1;
           else do;
              lastbinnum = z-1;
              distance_left = pmean - psums[z-1];
              leave;
           end;
        end;

*** Avoid loop for other (larger) p-value.
    If sumpvar equals last bin, p_right = 1 - p_left + lastbinsize.
    Otherwise, p_right = 1 - p_left.
***;
        if sumpvar = psums[lastbinnum] then do;
           lastbinsize=1;
           do k=lastbinnum to 1 by -1;
              if psums[k]=psums[k-1] then lastbinsize+1;
              leave;
           end;
           p_right = num_psmps - p_left + lastbinsize;
        end;
        else p_right = num_psmps - p_left;
     end;

     else do;
        do z=num_psmps to 1 by -1;
           if sumpvar<=psums[z] then p_right+1;
           else do;
              lastbinnum = z+1;
              distance_right = psums[z+1] - pmean;
              leave;
           end;
        end;

*** Avoid loop for other (larger) p-value.
    If psum equals last bin, p_left = 1 - p_right + lastbinsize.
    Otherwise, p_left = 1 - p_right.
***;
        if sumpvar = psums[lastbinnum] then do;
           lastbinsize=1;
           do k=lastbinnum to num_psmps;
              if psums[k]=psums[k+1] then lastbinsize+1;
              else leave;
           end;
           p_left = num_psmps - p_right + lastbinsize;
        end;
        else p_left = num_psmps - p_right;
```

```
     end;

*** Base 2-sided p-value on distance from mean of last (i.e. least extreme) bin
    of smaller p-value.  This is common practice.
***;
     if p_left<p_right then do;
        p_both = p_left;
        do z=num_psmps to 1 by -1;
           if (psums[z] - pmean) >= distance_left then p_both+1;
           else leave;
        end;
     end;
     else if p_left>p_right then do;
        p_both = p_right;
        do z=1 to num_psmps;
           if (pmean - psums[z]) >= distance_right then p_both+1;
           else leave;
        end;
     end;
     else p_both=num_psmps;

*** Account for possibility, due to psum=a particular bin value, that
    p_both>num_psmps.
***;
     p_both = min(p_both,num_psmps);
        END;

     p_left  = p_left  / num_psmps;
     p_right = p_right / num_psmps;
     p_both  = p_both  / num_psmps;

*** If CONTROL sample is smaller than TEST (which is atypical), reverse
*** p-values, as empirical distribution is mirror of itself.;

     if "C"=COMPRESS(UPCASE(scan("&samp2var.", byval_counter,' ')),' ') then do;
           hold = p_left;
           p_left = p_right;
           p_right = hold;
     end;

     label permvar   = "Permuted Variable"
           n_psamp   = "Size of Permutation Samples"
           num_psmps = "# of Permutation Samples"
           p_left    = "Left p-value"
           p_right   = "Right p-value"
           p_both    = "Two-Tailed p-value"
           ;
      output &outdata.;
  END;
  run;

*** Optional ***;
* proc datasets lib=work memtype=data kill nodetails;
*   run;

%MEND BEBB_SIM;

%BEBB_SIM(num_psmps = 1000,
          indata    = MFPTUS.pricing_data_2strata_100000,
          outdata   = MFPTUS.BEBB_SIM_100000_2strata,
          byvars    = geography segment, samp2var=cntrl_test,
```

```
          permvar   = price,
          seed      =
           );
```

Appendix B

For the convenience of the reader, a proof is provided below for the validity of the SRSWOR procedure used by OPDN, which is essentially the approach of Goodman & Hedetniemi (1977). The algorithm as implemented in OPDN is shown again below.

OPDN implementation of Goodman & Hedetniemi (1977) for Permutation Tests:

*** temp[] is the array filled with all the data values, for current stratum, of the variable being permuted
*** psums[] is the array containing the permutation sample statistic values for every permutation sample

```
do m = 1 to #permutation tests
   x ← 0
   tot_FREQ_hold ← # records in current stratum
   tot_FREQ ← tot_FREQ_hold
   do n = 1 to # records in smaller of Control and Treatment samples
      cell ← uniform random variate on 1 to tot_FREQ
      x    ← temp[cell] + x
      hold ← temp[cell]
      temp[cell] ← temp[tot_FREQ]
      temp[tot_FREQ] ← hold
      tot_FREQ ← tot_FREQ -1
   end;
   psums[m] ← x
end;
```

For a sampling algorithm to be a valid SRSWOR procedure, the probability of selecting any without-replacement sample of n items from a population of N items ($N > n$) needs to equal the probability of selecting any other without-replacement sample of n items, and that probability is:

Pr(drawing any particular sample of n items from larger group of N items) = $1 \Big/ \binom{N}{n} = n!(N-n)!/N!$ (C1)

because there are N – choose – n possible without-replacement samples.

Using the algorithm shown above, the probability of drawing the first item is $1/N$, and the probability of drawing the second item is $1/(N\text{-}1)$ and the probability of drawing the third item is $1/(N\text{-}2)$, and so on. Because each of these draws is independent of the others, the probability of drawing a sample of any n items is the product of these probabilities:

Pr(drawing any particular sample of n items from larger group of N items) =

$\frac{1}{N} \cdot \frac{1}{(N-1)} \cdot \frac{1}{(N-2)} \cdot \ldots \cdot \frac{1}{(N-n+2)} \cdot \frac{1}{(N-n+1)}$ * the number of permutations of these n items (because we do not care about the ordering of the n items, only that a particular set of n items is drawn), which is $n!$.

So Pr(drawing any particular sample of *n* items from larger group of *N* items) = $n! \Big/ \prod_{i=0}^{n-1}(N-i)$ (B2)

But in fact, (B1) = (B2), as shown below.

(B1) = $\frac{n!(N-n)!}{N!} = \frac{[n(n-1)(n-2)\cdot\ldots\cdot 2\cdot 1][(N-n)(N-n-1)(N-n-2)\cdot\ldots\cdot 2\cdot 1]}{N(N-1)(N-2)\cdot\ldots\cdot(N-n+1)(N-n)(N-n-1)\cdot\ldots\cdot 2\cdot 1} =$

$= \frac{[n(n-1)(n-2)\cdot\ldots\cdot 2\cdot 1]}{N(N-1)(N-2)\cdot\ldots\cdot(N-n+1)} = \frac{n!}{\prod_{i=0}^{n-1}(N-i)} =$

= (B2)

Appendix C

SAS® v.9.2 code for the OPDY, PSS, A4.8, DA, and Out-SM algorithms, and the SAS® code that generates the datasets used to test them in this paper, is presented below.

```
************************************************************************
************************************************************************
PROGRAM:  MFBUS.SAS

DATE:     9/13/10

CODER:    J.D. Opdyke

PURPOSE:  Run and compare different SAS bootstrap algorithms including OPDY, PSS, DA,
          Out-SM, and A4.8.  See Opdyke, J.D., "Much Faster Bootstraps Using SAS,"
          InterStat, October, 2101, for detailed explanations.

INPUTS:   Each macro is completely modular and accepts 5 macro parameters:
          bsmp_size = number of observations in each of the bootstrap samples
          num_bsmps = number of bootstrap samples
          indata    = the input dataset (including libname)
          byvars    = the "by variables" defining the strata
          bootvar   = the variable to be bootstrapped

OUTPUTS:  A uniquely named SAS dataset, the name of which contains the name of the
          algorithm, bsmp_size, and num_bsmps.  Variables in the output dataset include
          the mean of the bootstrap statistics, the standard deviation of the bootstrap
          statistics, and the 2.5th and 97.5th percentiles of the bootstrap statistics.
          Additional or different bootstrap statistics are easily added.

CAVEATS:  The "by variables" defining the strata in the input datasets are, in OPDY and
          DA, assumed to be character variables.
```

```
          The directory c:\MFBUS must be created before the program is run.

*************************************************************************
*************************************************************************
***;


options
label
symbolgen
fullstimer
yearcutoff=1950
nocenter
ls = 256
ps = 51
msymtabmax=max
mprint
mlogic
minoperator mindelimiter=' '
cleanup
;


libname MFBUS "c:\MFBUS";


%macro makedata(strata_size=, numsegs=, numgeogs=);

%let numstrata = %eval(&numsegs.*&numgeogs.);

*** For the price variable, multiplying the random variates by the loop counter
    dramatically skews the values of the sample space, thus ensuring that any
    erroneous non-random sampling will be spotted quickly and easily.
***;

data MFBUS.price_data_&numstrata.strata_&strata_size.(keep=geography segment price
sortedby=geography segment);
  format segment geography $8.;
  array seg{3} $ _TEMPORARY_ ('segment1' 'segment2' 'segment3');
  array geog{4} $ _TEMPORARY_ ('geog1' 'geog2' 'geog3' 'geog4');
  strata_size = 1* &strata_size.;
  do x=1 to &numgeogs.;
     geography=geog{x};
     do j=1 to &numsegs.;
        segment=seg{j};
        if j=1 then do i=1 to strata_size;
           price=rand('UNIFORM')*10*i;
           output;
        end;
        else if j=2 then do i=1 to strata_size;
           price=rand('NORMAL')*10*i;
           output;
        end;
        else if j=3 then do i=1 to strata_size;
           price=rand('LOGNORMAL')*10*i;
           output;
        end;
     end;
  end;
  run;
```

```
%mend makedata;


%makedata(strata_size=10000, numsegs=2, numgeogs=1);
%makedata(strata_size=10000, numsegs=2, numgeogs=3);
%makedata(strata_size=10000, numsegs=3, numgeogs=4);

%makedata(strata_size=100000, numsegs=2, numgeogs=1);
%makedata(strata_size=100000, numsegs=2, numgeogs=3);
%makedata(strata_size=100000, numsegs=3, numgeogs=4);

%makedata(strata_size=1000000, numsegs=2, numgeogs=1);
%makedata(strata_size=1000000, numsegs=2, numgeogs=3);
%makedata(strata_size=1000000, numsegs=3, numgeogs=4);

%makedata(strata_size=10000000, numsegs=2, numgeogs=1);
%makedata(strata_size=10000000, numsegs=2, numgeogs=3);
%makedata(strata_size=10000000, numsegs=3, numgeogs=4);
```

***** OPDY_Boot ***;**
***** OPDY_Boot ***;**
***** OPDY_Boot ***;**

```
%macro OPDY_Boot(bsmp_size=, num_bsmps=, indata=, byvars=, bootvar=);


*** the only assumption made within this macro is that the byvars are all character
variables;


*** obtain last byvar, count byvars, and assign each byvar into macro variables for
easy access/processing;

%let last_byvar = %scan(&byvars.,-1);
%let num_byvars = %sysfunc(countw(&byvars.));
%do i=1 %to &num_byvars.;
  %let byvar&i. = %scan(&byvars.,&i.);
%end;


*** macro obtains number of observations in a dataset;

%macro nobs(dset);
   %if %sysfunc(exist(&dset)) %then %do;
      %let dsid=%sysfunc(open(&dset));
      %let nobs=%sysfunc(attrn(&dsid,nobs));
      %let dsid=%sysfunc(close(&dsid));
   %end;
   %else %let nobs=0;
   &nobs
%mend nobs;


*** initialize macro variables used later;
%let bmean =;
%let bstd =;
%let b975 =;
```

```
%let b025 =;


*** obtain counts and cumulated counts for each strata;

proc summary data=&indata. nway;
  class &byvars.;
  var &bootvar.;
  output out=byvar_nobs(keep=_FREQ_ &byvars.) n=junk;
  run;

%let n_byvals = %nobs(byvar_nobs);

data cum_temp(keep=_FREQ_ cum_prev_freq);
  set byvar_nobs(keep=_FREQ_);
  retain cum_prev_freq 0;
  prev_freq = lag(_FREQ_);
  if _n_=1 then prev_freq = 0;
  cum_prev_freq = sum(cum_prev_freq, prev_freq);
  run;


*** put counts, cumulated counts, and byvar values into macro strings;

proc sql noprint;
  select cum_prev_freq into :cum_prev_freqs separated by ' ' from cum_temp;
  quit;

proc sql noprint;
  select _freq_ into :freqs separated by ' ' from cum_temp;
  quit;

%do i=1 %to &num_byvars.;
   proc sql noprint;
     select &&byvar&i. into :byvals&i. separated by ' ' from byvar_nobs;
     quit;
%end;


*** get size of largest stratum;

proc summary data=byvar_nobs(keep=_FREQ_) nway;
  var _FREQ_;
  output out=byvar_nobs(keep=max_freq) max=max_freq;
  run;
data _null_;
  set byvar_nobs;
  call symputx('max_freq',max_freq);
  run;


*** save results of each stratum in cumulated macro variables instead of outputting to
a dataset on the data step to lessen intermediate memory requirements
***;


data _null_;
  set &indata.(keep=&byvars. &bootvar.);
  by &byvars.;
  array temp{&max_freq.} _TEMPORARY_;
```

```
  %if &sysver >= 9.2 %then %do;
    array bmeans{&num_bsmps.} _TEMPORARY_;
    retain byval_counter 0 cum_prev_freq 0;
  %end;
  %if &sysver < 9.2 %then %do;
    array bmeans {&num_bsmps.} bmean1-bmean&num_bsmps.;
    retain byval_counter 0 cum_prev_freq 0 bmean1-bmean&num_bsmps.;
  %end;
  temp[_n_-cum_prev_freq]=&bootvar.;
  if last.&last_byvar. then do;
     byval_counter+1;
     freq = 1* scan("&freqs.", byval_counter,' ');
     num_bsmps = &num_bsmps.*1;
     bsmp_size = &bsmp_size.*1;
     do m=1 to num_bsmps;
        x=0;
        do n=1 to bsmp_size;
           x = temp[floor(ranuni(-1)*freq) + 1] + x;
        end;
        bmeans[m] = x/bsmp_size;
     end;
     bmean = mean(of bmeans[*]);
     bstd = std(of bmeans[*]);
     b975 = pctl(97.5, of bmeans[*]);
     b025 = pctl(2.5, of bmeans[*]);
     call symput('bmean',symget('bmean')||" "||compress(bmean));
     call symput('bstd',symget('bstd')||" "||compress(bstd));
     call symput('b975',symget('b975')||" "||compress(b975));
     call symput('b025',symget('b025')||" "||compress(b025));
     cum_prev_freq = 1*scan("&cum_prev_freqs.",byval_counter+1,' ');
  end;
  run;


*** obtain and assign the format of each byvar, all of which are assumed to be
character variables;

data lens(keep=lens);
  set &indata.(keep=&byvars. firstobs=1 obs=1);
  do i=1 to &num_byvars.;
     lens = vlengthx(scan("&byvars.",i));
     output;
  end;
  run;
proc sql noprint;
  select lens into :alllens separated by ' ' from lens;
  quit;
%macro assign_formats;
   %do i=1 %to &num_byvars.;
      &&byvar&i. $%scan(&alllens.,&i.).
   %end;
%mend assign_formats;


*** assign each byvar value for each stratum;

%macro assign_byvar_vals(which_strata=);
   %do j=1 %to &num_byvars.;
      &&byvar&j. = scan("&&byvals&j.",&which_strata.,' ');
   %end;
%mend assign_byvar_vals;
```

```
*** unwind and assign all the cumulated macro variables;

data MFBUS.OPDY_boots_&bsmp_size._&num_bsmps.(sortedby=&byvars. drop=n_byvals i);
  n_byvals = 1*&n_byvals.;
  format %assign_formats;
  do i=1 to n_byvals;
     bmean = 1*scan("&bmean.",i,' ');
     bstd = 1*scan("&bstd.",i,' ');
     b025 = 1*scan("&b025.",i,' ');
     b975 = 1*scan("&b975.",i,' ');
     %assign_byvar_vals(which_strata = i)
     output;
  end;
  run;


*** optional ***;
* proc datasets lib=work memtype=data kill nodetails;
*   run;

%mend OPDY_Boot;


%OPDY_Boot(bsmp_size=500,
           num_bsmps=500,
           indata=MFBUS.price_data_6strata_100000,
           byvars=geography segment,
           bootvar=price
          );


*** PSS ***;
*** PSS ***;
*** PSS ***;

%macro PSS(bsmp_size=, num_bsmps=, indata=, byvars=, bootvar=);

proc surveyselect data=&indata. method=urs sampsize=&bsmp_size. rep=&num_bsmps. seed=-1
out=Boot_PSS_Samps(drop=expectedhits samplingweight) noprint;
  strata &byvars.;
  run;

proc summary data=Boot_PSS_Samps nway;
  class &byvars. replicate;
  weight numberhits;
  var &bootvar.;
  output out=Boot_PSS_avgs(sortedby=&byvars. keep=&byvars. &bootvar.) mean=;
  run;

proc univariate data=Boot_PSS_avgs noprint;
  by &byvars.;
  var &bootvar.;
  output out=MFBUS.Boot_PSS_&bsmp_size._&num_bsmps.
        mean=bmean
         std=bstd
    pctlpts = 2.5 97.5
     pctlpre=b
```

```
  ;
  run;


*** optional;
* proc datasets lib=work memtype=data kill nodetails;
*   run;

%mend PSS;


%PSS(bsmp_size=500,
     num_bsmps=500,
     indata=MFBUS.price_data_6strata_100000,
     byvars=geography segment,
     bootvar=price
    );
```

***** Boot_DA ***;**
***** Boot_DA ***;**
***** Boot_DA ***;**

```
%macro Boot_DA(bsmp_size=, num_bsmps=, indata=, byvars=, bootvar=);

*** the only assumption made within this macro is that the byvars are all character
variables;

*** obtain last byvar, count byvars, and assign each byvar into macro variables for
easy access/processing;

%let last_byvar = %scan(&byvars.,-1);
%let num_byvars = %sysfunc(countw(&byvars.));
%do i=1 %to &num_byvars.;
  %let byvar&i. = %scan(&byvars.,&i.);
%end;


*** macro obtains number of observations in a dataset;

%macro nobs(dset);
   %if %sysfunc(exist(&dset)) %then %do;
      %let dsid=%sysfunc(open(&dset));
      %let nobs=%sysfunc(attrn(&dsid,nobs));
      %let dsid=%sysfunc(close(&dsid));
   %end;
   %else %let nobs=0;
   &nobs
%mend nobs;


*** obtain counts and cumulated counts for each strata;

proc summary data=&indata. nway;
  class &byvars.;
  var &bootvar.;
  output out=byvar_nobs(keep=_FREQ_ &byvars.) n=junk;
  run;
```

```
%let n_byvals = %nobs(byvar_nobs);

data cum_temp(keep=_FREQ_ cum_prev_freq);
  set byvar_nobs(keep=_FREQ_);
  retain cum_prev_freq 0;
  prev_freq = lag(_FREQ_);
  if _n_=1 then prev_freq = 0;
  cum_prev_freq = sum(cum_prev_freq, prev_freq);
  run;


*** put counts, cumulated counts, and byvar values into macro strings;

proc sql noprint;
  select cum_prev_freq into :cum_prev_freqs separated by ' ' from cum_temp;
  quit;
proc sql noprint;
  select _freq_ into :freqs separated by ' ' from cum_temp;
  quit;

%do i=1 %to &num_byvars.;
   proc sql noprint;
     select &&byvar&i. into :byvals&i. separated by ' ' from byvar_nobs;
     quit;
%end;


*** obtain and assign the format of each byvar, all of which are assumed to be
character variables;

data lens(keep=lens);
  set &indata.(keep=&byvars. firstobs=1 obs=1);
  do i=1 to &num_byvars.;
     lens = vlengthx(scan("&byvars.",i));
     output;
  end;
  run;
proc sql noprint;
  select lens into :alllens separated by ' ' from lens;
  quit;
%macro assign_formats;
   %do i=1 %to &num_byvars.;
      &&byvar&i. $%scan(&alllens.,&i.).
   %end;
%mend assign_formats;


*** assign each byvar value for each stratum;

%macro assign_byvar_vals(which_strata=);
   %do j=1 %to &num_byvars.;
      &&byvar&j. = scan("&&byvals&j.",&which_strata.,' ');
   %end;
%mend assign_byvar_vals;


data MFBUS.boot_da_&bsmp_size._&num_bsmps.(keep=&byvars. bmean bstd b975 b025);
  n_byvals=&n_byvals.;
  bsmp_size = 1* &bsmp_size.;
  num_bsmps = 1* &num_bsmps.;
  format %assign_formats;
```

```
  do byval_counter=1 to n_byvals;
     freq          = 1* scan("&freqs.", byval_counter,' ');
     cum_prev_freq = 1* scan("&cum_prev_freqs.", byval_counter,' ');
     %assign_byvar_vals(which_strata = byval_counter)
     array bmeans{&num_bsmps.} bm1-bm&num_bsmps. (&num_bsmps.*0);
     do bsample=1 to num_bsmps;
        xsum=0;
        do obs=1 to bsmp_size;
           obsnum = floor(freq*ranuni(-1))+1+cum_prev_freq;
           set &indata.(keep=&bootvar.) point=obsnum;
           xsum = xsum + &bootvar.;
        end;
        bmeans[bsample] = xsum/bsmp_size;
     end;
     bmean = mean(of bm1-bm&num_bsmps.);
     bstd = std(of bm1-bm&num_bsmps.);
     b025 = pctl(2.5, of bm1-bm&num_bsmps.);
     b975 = pctl(97.5, of bm1-bm&num_bsmps.);
     output;
  end;
  stop;
  run;


*** optional ***;
* proc datasets lib=work memtype=data kill nodetails;
*   run;

%mend Boot_DA;


%Boot_DA(bsmp_size=500,
         num_bsmps=500,
         indata=MFBUS.price_data_6strata_100000,
         byvars=geography segment,
         bootvar=price
        );
```

***** Boot_SM ***;**
***** Boot_SM ***;**
***** Boot_SM ***;**

```
%macro Boot_SM(bsmp_size=, num_bsmps=, indata=, byvars=, bootvar=);

*** obtain counts by strata;

proc summary data=&indata. nway;
  class &byvars.;
  var &bootvar.;
  output out=byvar_nobs(keep=_FREQ_ &byvars.) n=junk;
  run;

*** output bootstrap observations to sample in a nested loop;

data bsmp;
  set byvar_nobs(keep=&byvars. _FREQ_);
  num_bsmps = 1*&num_bsmps.;
  bsmp_size = 1*&bsmp_size.;
```

```
  do sample=1 to num_bsmps;
     do k=1 to bsmp_size;
        obsnum = floor(_FREQ_*ranuni(-1))+1;
        output;
     end;
  end;
  run;

proc sort data=bsmp;
  by &byvars. obsnum;
  run;

proc sort data=&indata. out=price_data;
  by &byvars.;
  run;


*** create record counter on input dataset;

%let last_byvar = %scan(&byvars.,-1);

data boot;
  set price_data;
  retain obsnum 0;
  by &byvars.;
  if first.&last_byvar. then obsnum=1;
  else obsnum+1;
  run;

proc sort data=boot;
  by &byvars. obsnum;
  run;


*** merge bootstrap sample observations with input dataset to obtain bootstrap samples;

data boot
     error
        ;
  merge boot(in=inboot)
        bsmp(in=inbsmp)
        ;
  by &byvars. obsnum;
  if inboot & inbsmp then output boot;
  else if inboot~=1 then output error;
  run;


*** summarize bootstrap samples;

proc summary data=boot(keep=&byvars. sample &bootvar.) nway;
  class &byvars. sample;
  var &bootvar.;
  output out=boot_means(keep=&byvars. &bootvar. sortedby=&byvars.) mean=;
  run;

proc univariate data=boot_means(keep=&byvars. &bootvar.) noprint;
  by &byvars.;
  var &bootvar.;
  output out=MFBUS.boot_Out_SM_&bsmp_size._&num_bsmps.
                                  mean = b_mean
```

```
                                    std = b_std
                                    pctlpts = 2.5 97.5 pctlpre=b
                                    ;
      run;

*** optional ***;
* proc datasets lib=work memtype=data kill nodetails;
*   run;

%mend Boot_SM;

%Boot_SM(bsmp_size=500,
         num_bsmps=500,
         indata=MFBUS.price_data_6strata_100000,
         byvars=geography segment,
         bootvar=price
        );
```

***** ALGO4p8 ***;**
***** ALGO4p8 ***;**
***** ALGO4p8 ***;**

```
%macro Algo4p8(bsmp_size=, num_bsmps=, indata=, byvars=, bootvar=);

*** obtain counts by strata and merge onto input dataset;

proc summary data=&indata. nway;
  class &byvars.;
  var &bootvar.;
  output out=byvar_nobs(keep=_FREQ_ &byvars.) n=junk;
  run;

proc sort data=&indata. out=price_data;
  by &byvars.;
  run;

data price_data
     error
         ;
  merge byvar_nobs(in=innobs keep=&byvars. _FREQ_)
        &indata.(in=infull)
              ;
  by &byvars.;
  if innobs & infull then output price_data;
  else output error;
  run;


*** simultaneously create all bootstrap samples and summarize results of each as the
end of stratum is reached;

%let last_byvar = %scan(&byvars.,-1);

data MFBUS.boot_algo4p8_&bsmp_size._&num_bsmps.(keep=&byvars. bstd bmean b025 b975);
  set price_data end=lastrec;
  by &byvars.;
  num_bsmps = &num_bsmps.;
  array bsummeans{&num_bsmps.} bsummean_1-bsummean_&num_bsmps.;
```

```
  array bcds{&num_bsmps.} bcd_1-bcd_&num_bsmps.;
  retain bsummean_1-bsummean_&num_bsmps. 0 bcd_1-bcd_&num_bsmps. &bsmp_size. counter 0;
  counter+1;
  p = 1/(_FREQ_-counter+1);
  do i=1 to num_bsmps;
     if bcds[i]>0 then do;
        x = rand('BINOMIAL',p,bcds[i]);
        bsummeans[i]=x*&bootvar. + bsummeans[i];
        bcds[i]=bcds[i]-x;
     end;
  end;
  if last.&last_byvar. then do;

     bsmp_size = 1*&bsmp_size.;
     do h=1 to num_bsmps;
        bsummeans[h] = bsummeans[h]/bsmp_size;
     end;
     bmean = mean(of bsummean_1-bsummean_&num_bsmps.);
     bstd = std(of bsummean_1-bsummean_&num_bsmps.);
     b025 = pctl(2.5, of bsummean_1-bsummean_&num_bsmps.);
     b975 = pctl(97.5, of bsummean_1-bsummean_&num_bsmps.);
     output;
     if lastrec~=1 then do;
        counter = 0;
        do x=1 to num_bsmps;
           bsummeans[x]=0;
           bcds[x]=bsmp_size;
        end;
     end;
  end;
  run;

*** optional;
* proc datasets lib=work memtype=data kill nodetails;
*   run;

%mend Algo4p8;

%Algo4p8(bsmp_size=500,
         num_bsmps=500,
         indata=MFBUS.price_data_6strata_100000,
         byvars=geography segment,
         bootvar=price
        );
```

Appendix D

To prove the validity of Algorithm 4.8 as an equal-probability, with-replacement sampling algorithm, it must be shown that all possible samples of *n* items have equal probability of being selected. The probability of drawing any specific item in any draw from among the *N* items is 1/*N*, so the probability of a particular set of *n* items being drawn, in a specific order, is simply $(1/N)^n$ or $1/N^n$.[15] However, the order of selection does

[15] This corresponds with there being N^n possible sample-orderings for a with-replacement sample of *n* items drawn from *N* items, $n \leq N$.

not matter for these purposes,[16] so we must use the probability of drawing a particular sample of *n* items, in any order, and show that this is identical to the probability of drawing any sample of *n* items using Algo4.8.

The probability of drawing any particular sample of *n* items, in any order, when sampling with replacement is given by Efron and Tibshirani (1993, p. 58) as

$$\frac{n!}{b_1!b_2!...b_n!}\cdot\prod_{i=1}^{n}\binom{1}{n}^{b_i} \tag{D1}$$

when $n = N$, and b_i indicates the number of times item *i* is drawn. Note that, by definition, $n=\sum_{i=1}^{n} b_i$, so $\prod_{i=1}^{n}(1/n)^{b_i} = (1/n)^{n}$. So when $n \le N$, (D1) is simply

$$\frac{n!}{b_1!b_2!...b_N!}\cdot\left(\frac{1}{N}\right)^{n} \tag{D2}$$

To show that the probability that any sample of *n* items drawn from *N* items ($n \le N$) using Algo4.8 is equal to (D2), note that the probability that any of the *n* items in the sample is drawn *b* times, where *b* is 0 ≤ positive integers ≤ *n'*, is by definition the binomial probability (using *n'* and *p* as defined in Algo4.8)

$$\Pr\left(b_i = b_i^*\right) = \binom{n'}{b_i^*} p^{b_i^*}\left(1-p\right)^{n'-b_i^*} \quad \text{for } 0 < p < 1 \text{ [17]} \tag{D3}$$

This makes the probability of drawing any *n* items with Algo4.8, with possible duplicates, in any order, the following:

[16] Note that given the sequential nature of the algorithm, the order of the *n* items will be determined by, and is the same as, the order of the *N* population items, which of course can be sorted in any way without affecting the validity of the Algo4.8 algorithm.

[17] In Algo4.8, as mentioned above, *p* will never equal zero, and if *p* = 1, $b_i^* = n'$, which is correct.

Pr(Algo4.8 sample = any particular with-replacement sample) =

$$\frac{n!}{b_1!(n-b_1)!}\left(\frac{1}{N}\right)^{b_1}\left(1-\frac{1}{N}\right)^{n-b_1}\cdot$$

$$\frac{(n-b_1)!}{b_2!(n-b_1-b_2)!}\left(\frac{1}{N-1}\right)^{b_2}\left(1-\frac{1}{N-1}\right)^{n-b_1-b_2}\cdot$$

$$\frac{(n-b_1-b_2)!}{b_3!(n-b_1-b_2-b_3)!}\left(\frac{1}{N-2}\right)^{b_3}\left(1-\frac{1}{N-2}\right)^{n-b_1-b_2-b_3}\cdot$$

$$\vdots$$

$$\frac{(n-b_1-b_2-\ldots-b_{N-1})!}{b_N!(n-b_1-b_2-\ldots-b_N)!}\left(\frac{1}{N-(N-1)}\right)^{b_N}\left(1-\frac{1}{N-(N-1)}\right)^{n-b_1-b_2-b_3-\ldots-b_N} \quad \text{(D4)}$$

Reordering terms gives

Pr(Algo4.8 sample = any particular with-replacement sample) =

$$\frac{n!}{b_1!(n-b_1)}\cdot\frac{(n-b_1)!}{b_2!(n-b_1-b_2)}\cdot\frac{(n-b_1-b_2)!}{b_3!(n-b_1-b_2-b_3)}\cdot\ldots\cdot\frac{(n-b_1-b_2-\ldots-b_{N-1})!}{b_N!(n-b_1-b_2-\ldots-b_N)!}\cdot$$

$$\left(\frac{1}{N}\right)^{b_1}\left(\frac{N-1}{N}\right)^{n-b_1}\cdot\left(\frac{1}{N-1}\right)^{b_2}\left(\frac{N-2}{N-1}\right)^{n-b_1-b_2}\cdot\left(\frac{1}{N-2}\right)^{b_3}\left(\frac{N-3}{N-2}\right)^{n-b_1-b_2-b_3}\cdot\ldots\cdot$$

$$\left(\frac{1}{N-(N-1)}\right)^{b_N}\left(\frac{N-N}{N-(N-1)}\right)^{n-b_1-b_2-b_3-\ldots-b_N} \quad \text{(D5)}$$

Because $n=\sum_{i=1}^{n} b_i$, the denominator in the last "combinatoric" term in (D5) is $b_N!0!=b_N!$, and except for the first numerator *n*! and b_i! in each denominator, the rest of the numerators and denominators in the combinatoric terms cancel leaving $n!/b_1!b_2!\ldots b_N!$. Of the remaining "probability" terms, the final term can be written as

$$\left(\frac{1}{N-(N-1)}\right)^{b_N}\left(\frac{1}{N-(N-1)}\right)^{n-b_1-b_2-b_3-\ldots-b_N}\left(\frac{0}{1}\right)^{0}$$. If we avoid the centuries old debate (at least since the time of Euler) regarding the value of 0^0 and define $0^0 = 1$, as is accepted convention by numerous august mathematicians (including Euler),[18] then all the (N - x) numerators and denominators cancel here as well, leaving only, from the first term, $(1/N)^{b_1}(1/N)^{n-b_1} = (1/N)^n$, which yields

$$\frac{n!}{b_1!b_2!\ldots b_N!}\cdot\left(\frac{1}{N}\right)^n$$, which is (D2).

Excel workbook examples of calculating this probability using both (D2) directly and the steps of Algo4.8, for both $n \leq N$ and $n = N$, are available from the author upon request.

References

[1] Bebbington, A. (1975), "A Simple Method of Drawing a Sample Without Replacement," *Journal of the Royal Statistical Society, Series C (Applied Statistics)*, Vol. 24, No. 1, 136.

[2] Bentley, J. L. and Floyd, R. (1987), "A Sample of Brilliance," *Communications of the Association for Computing Machinery*, 30, 754–757.

[3] Bentley, J. L. and Knuth, D. (1986), "Literate Programming," *Communications of the Association for Computing Machinery*, 29, 364–369.

[4] Cassell, D. (2007), "Don't Be Loopy: Re-Sampling and Simulation the SAS Way," Proceedings of the SAS Global Forum 2007 Conference, Cary, NC: SAS Institute Inc.

[5] Chernick, M. (2007), *Bootstrap Methods: A Guide for Practitioners and Researchers*, 2nd ed., Hoboken, NJ, John Wiley & Sons, Inc.

[6] Davison, A., and Hinkley, D. (1997), *Bootstrap Methods and their Application*, Cambridge, UK, Cambridge University Press.

[7] Edgington, E.S., and Onghena, P. (2007), Randomization Tests, Fourth Edition, Chapman & Hall/CRC.

[8] Efron, B. (1979), "Bootstrap Methods: Another Look at the Jackknife," *Annals of Statistics*, 7, 1–26.

[9] Efron, B., and Tibshirani, R. (1993), *An Introduction to the Bootstrap*, New York, Chapman & Hall, LLC.

[10] Ernvall, J., & O. Nevalainen (1982), "An Algorithm for Unbiased Random Sampling," *The Computer Journal*, Vol.25 (1), p.45-47.

[11] Euler, L. (1748), translated by J.D. Blanton (1988), *Introduction to Analysis of the Infinite*, New York, NY, Springer-Verlag.

[12] Euler, L. (1770), translated by Rev. John Hewlett (1984), *Elements of Algebra*, New York, NY, Springer-Verlag.

[13] Fan, C. T., Muller, M. E., and Rezucha, I. (1962), "Development of Sampling Plans by Using Sequential (Item by Item) Selection Techniques and Digital Computers, " *Journal of the American Statistical Association*, 57, 387–402.

[18] See Euler (1748), Euler (1770), Graham et al., (1994), Knuth (1992), and Vaughan (1970).

[14] Fisher, Sir R.A. (1935), *Design of Experiments*, Edinburgh, Oliver & Boyd.
[15] Goodman, S. & S. Hedetniemi (1977), *Introduction to the Design and Analysis of Algorithms*, McGraw-Hill, New York.
[16] Good, P. (2004), Permutation, Parametric, and Bootstrap Tests of Hypotheses, Third Edition, Springer Verlag.
[17] Graham, R., Knuth, D., and Patashnik, O. (1994), *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed., Reading, MA: Addison-Wesley Publishing Company.
[18] Knuth, D. (May, 1992), "Two Notes on Notation," *The American Mathematical Monthly*, Vol. 99, No. 5.
[19] Manly, B. F. J., (2007), Randomization, Bootstrap and Monte Carlo Methods in Biology, Third Edition, Chapman & Hall/CRC.
[20] Mielke, P. & K. Berry (2001), *Permutation Methods: A Distance Function Approach*, Springer-Verlag, New York.
[21] Opdyke, J.D. (2010), "Much Faster Bootstraps Using SAS®," *InterStat*, October, 2010.
[22] Opdyke, J.D. (2011), "Permutation Tests (and Sampling Without Replacement) Orders of Magnitude Faster Using SAS®," *InterStat*, October, 2010.
[23] Pesarin, F. (2001), *Multivariate Permutation Tests with Applications in Biostatistics*, John Wiley & Sons, Ltd., New York.
[24] SAS Institute Inc. (2007), *SAS OnlineDoc 9.2*, Cary, NC: SAS Institute Inc.
[25] Secosky, J., and Bloom, J. (May, 2007), "Getting Started with the DATA Step Hash Object," SAS Institute, Inc.
[26] Secosky, J., and Bloom, J. (2008), "Better Hashing in SAS® 9.2," SAS Institute, Inc., Paper 306-2008.
[27] Tillé, Y. (2006), *Sampling Algorithms*, New York, NY, Springer.
[28] Tillé, Y. (2010), *eMail Correspondence*, September 30, 2010.
[29] Vaughan, H. (February 1970), "The Expression of 0^0," *The Mathematics Teacher*, Vol. 63, p.111.

Related Articles

Article ID	Article title
098	Bootstrap
096	Resampling
551	Bootstrapping Regression