PharmaSUG 2019 - Paper BP-219

Making the Days Count: Counting Distinct Days in Overlapping or Disjoint Date Intervals

Noory Kim, Synteract

ABSTRACT

Suppose you want to combine multiple date intervals. How do you avoid double-counting calendar days when the intervals overlap? Previous papers on this topic have followed a sensible "combine-then-count" method in principle, but in practice have presented code implementations whose calculations are not easy to follow. This paper presents a macro implementation that follows the "combine-then-count" paradigm more closely for the purpose of having calculations that are easier to follow and verify.

INTRODUCTION

The statistical analysis plan (SAP) of a particular study called for determining the number of calendar days during study treatment when a subject also took at least one standard-of-care medication. For a calendar day when a subject took more than one standard-of-care medication, the SAP specified counting that day only once.

To count the number of distinct calendar days among multiple overlapping intervals, we can use a two-stage method similar to a method discussed by Cheng (2006). First, as illustrated in Figure 1, we combine the date intervals (at the top in blue) such that the resulting intervals (at the bottom in orange) are disjoint and do not double-count any calendar day. Second, we count up the number of days in the combined intervals. Since these combined intervals are disjoint, the calendar days they span will be distinct.

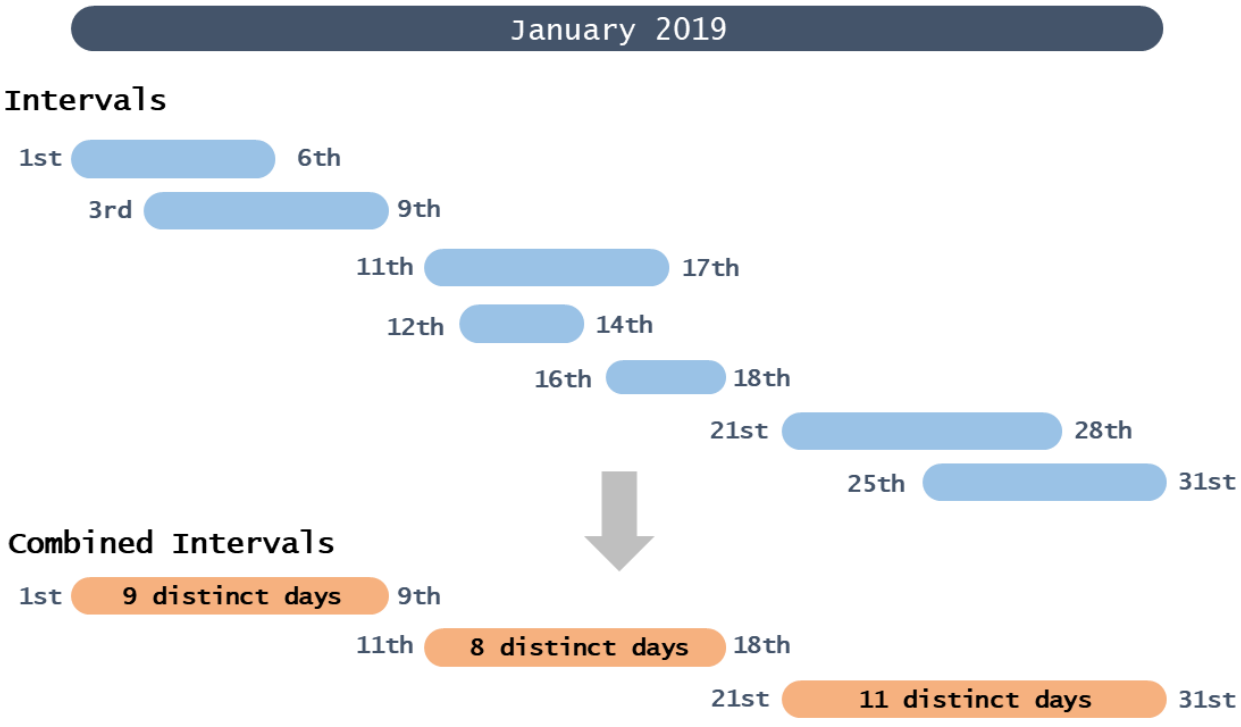

Figure 1. Date intervals (in blue) and combined intervals (in orange) are shown for subject 101-001 from the example discussed below. The cardinal numbers surrounding each interval denote the start and end dates of the interval. (e.g. For the first interval, "1st" to the left indicates 01JAN2019 as the start date, and "6th" to the right indicates 06JAN2019 as the end date.) The text inside each combined interval indicates the number of distinct calendar days within that interval.

In their SAS© user conference papers, Cheng (2006) and Su (2007) both present a variation of a "combine-then-count" approach. Cheng's implementation truncates intervals before combining them. Su's implementation keeps a running tally of the calendar days as it goes through the input data set row-by-row, in effect making pairwise but not global comparisons. With either implementation it is not readily apparent how the intervals get combined. As a result, their counts can be difficult to verify. Indeed, when run on the sample data set ONE (Table 2), they yield different results, as listed in Table 1.

Implementation	Output: Duration in Days for Subject 101-001
Cheng (2006), Method 1	28
Su (2007)	30

Table 1. Counts of distinct days for the sample data set ONE (Table 2 below) by other implementations.

As the results do not agree, it is evident that one of these implementations goes awry for this case. Manually counting the days in Figure 1 gives us a result of 28 (= 9+8+11), matching the result by Cheng. It is more difficult at first glance to determine what throws off Su's implementation and where.

In this paper I present an implementation that resembles the two stages of the "combine-then-count" method more closely for the purpose of having calculations that are easier to follow and verify.

MACRO DESIGN AND IMPLEMENTATION

ASSUMPTIONS ABOUT INPUT DATA

It is expected that the input data will have been prepped and cleaned to have the following properties:

1. The data set has an ID variable (such as subject ID) to denote which intervals go together. There are no missing values for ID.
2. There are no missing or partial date values. Each row in the data set will have a numeric (and thus complete) start date and a numeric (and complete) end date. All dates are actual calendar days (and exclude non-existent dates such as 31FEB2019).
3. All start dates are less than or equal to end dates.
4. The data does not include calendar days outside of the period of interest.

We will be using the following sample data set, named ONE (Table 2). This data is represented by the blue intervals in the top part of Figure 1.

USUBJID	STDT	ENDT
101-001	2019-01-01	2019-01-06
101-001	2019-01-03	2019-01-09
101-001	2019-01-11	2019-01-17
101-001	2019-01-12	2019-01-14
101-001	2019-01-16	2019-01-18
101-001	2019-01-21	2019-01-28
101-001	2019-01-25	2019-01-31

Table 2. The sample dataset, named ONE, used as the input data set for the example in this paper. USUBJID is the subject ID. STDT and ENDT are numeric variables with the date format yymmdd10.

PSEUDOCODE

The implementation in this paper will follow the two-stages of "combine-then-count":

Stage 0: Sort the input data

Stage 1: Combine the intervals into a set of disjoint intervals

Step 1.1: Determine which intervals overlap and can thus be combined.

Step 1.2: Determine the start and end date of each combined interval.

Stage 2: Count the number of distinct days

Step 2.1: Determine the number of (distinct) days in each combined interval.

Step 2.2: Add up the day counts from all combined intervals.

Note that the pseudocode specifies completing Stage 1 in its entirely before moving on to Stage 2, thereby adhering to a computer science design principle known as separation of concerns.

EXAMPLE

Let's walk through an example of how the macro works, using the sample data set ONE on the previous page as input. The macro call is as follows:

```
%countUniqueDays(inputDataset=one, idVariable=usubjid,
                 startDateVariable=stdt, endDateVariable=endt
                 outputDataset=two);
```

The first four macro parameters listed here are all required. The name of the output data set is specified here as TWO. (If outputDataset is not specified, then by default the output dataset will be named &inputDataset._daycount, which in this case would have been ONE_DAYCOUNT.)

A note on presentation: The code shown in the rest of the example will be shown with macro variables resolved to the values specified in this macro call. The macro parameters idVariable, startDateVarible, and endDateVariable will instead be referred to as "subject ID", "start date", and "end date" respectfully.

Stage 0: Sort the input data

Step 0: The macro first sorts the input data set by subject ID, start date, then end date.

```
proc sort data=one out=_temp_0;
  by usubjid stdt endt;
run;
```

This yields the temporary data set shown in Table 3. This is just a sorted subset of the input data set (Table 2). It includes the subject ID, start date, and end date.

USUBJID	STDT	ENDT
101-001	2019-01-01	2019-01-06
101-001	2019-01-03	2019-01-09
101-001	2019-01-11	2019-01-17
101-001	2019-01-12	2019-01-14
101-001	2019-01-16	2019-01-18
101-001	2019-01-21	2019-01-28
101-001	2019-01-25	2019-01-31

Table 3. The data set output by step 0. This is a sorted subset of the input data set, and includes the variables for subject ID, start date, and end date.

Stage 1: Combine the intervals into a set of disjoint intervals

Overview

In the first stage of the method, the macro combines the intervals. Figure 2 illustrates the steps the macro takes to do this. First, for each subject ID, the macro partitions the set of date intervals into subsets of intervals that each compose a continuous block of time. This step (1.1) is where the macro does the "heavy lifting" of accounting for overlaps. Then, after the intervals have been combined, the macro determines the start and end dates of each combined interval.

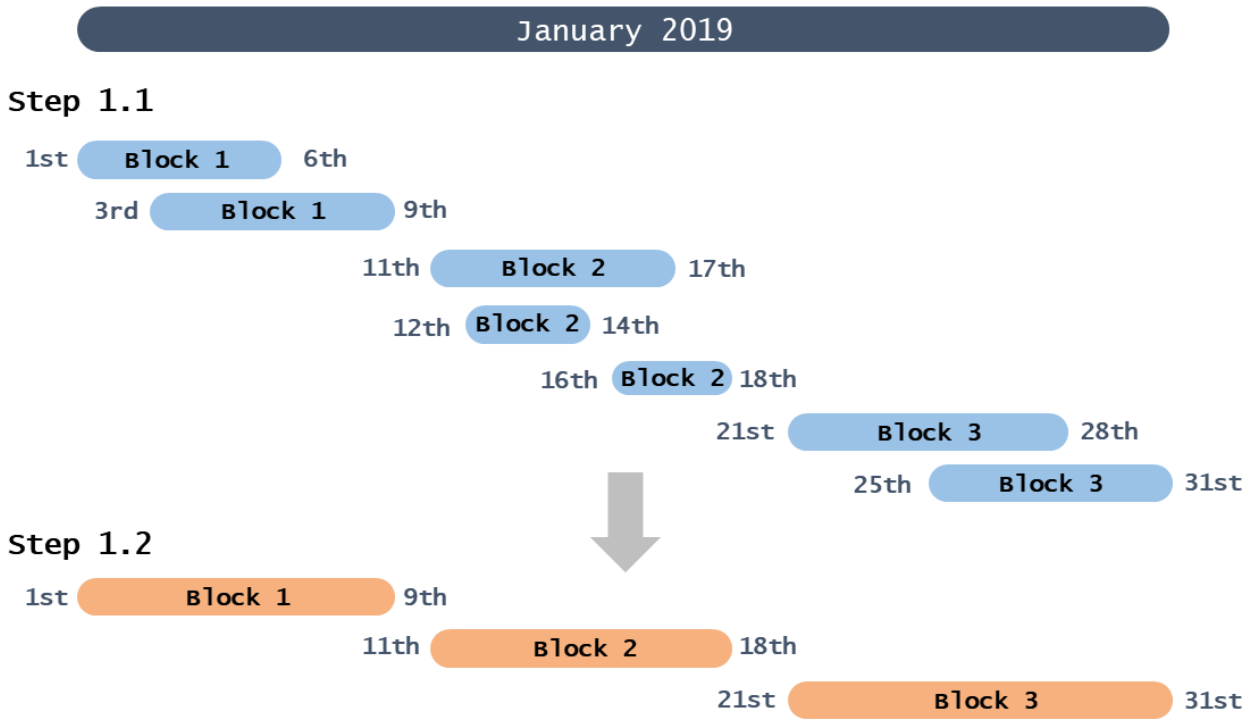

Figure 2. In the "combine" stage of the "combine-then-count" method, the macro first assigns BLOCKSEQ to partition the set of date intervals (for the same ID) into subsets of intervals that each compose a continuous block of time. (Step 1.1). Then the macro determines the start and end dates of each block. (Step 1.2)

Step 1.1: Determine which intervals overlap and can thus be combined.

The macro identifies which intervals can be combined into continuous interval.

Before we dig into the SAS code, let's first look at the output from this step, shown in Table 4.

Intervals that can be combined into a continuous block of time are assigned the same value of BLOCKSEQ. This is achieved by having BLOCKSEQ increment by one when the start date of a row (STDT) is more than one day after the lagging maximum end date (LAG_MAX_ENDT), i.e. when there is a gap in calendar days between intervals. LAG_MAX_ENDT is defined such that it is equal to the maximum value of the end date (ENDT) in all previous rows for the same subject ID. (LAG_MAX_ENDT is the lagging value of the maximum end date (MAX_ENDT), which is the maximum value of ENDT over all previous rows plus the current row for the same subject ID.)

USUBJID	BLOCKSEQ	LAG_MAX_ENDT	STDT	ENDT	MAX_ENDT	LAG_ENDT
101-001	1	.	2019-01-01	2019-01-06	2019-01-06	.
101-001	1	2019-01-06	2019-01-03	2019-01-09	2019-01-09	2019-01-06
101-001	2	2019-01-09	2019-01-11	2019-01-17	2019-01-17	2019-01-09
101-001	2	2019-01-17	2019-01-12	2019-01-14	2019-01-17	2019-01-17
101-001	2	**2019-01-17**	**2019-01-16**	2019-01-18	2019-01-18	**2019-01-14**
101-001	3	2019-01-18	2019-01-21	2019-01-28	2019-01-28	2019-01-18
101-001	3	2019-01-28	2019-01-25	2019-01-31	2019-01-31	2019-01-28

Table 4. The data set output by step 1.1, plus a column for the lagging end date (LAG_ENDT). BLOCKSEQ has the same value for individual intervals that can combine to form a continuous interval. BLOCKSEQ is incremented in rows where the start date (STDT) is more than one day after the "lagging maximum" end date (LAG_MAX_ENDT), which is the maximum end date (MAX_ENDT) over all previous rows for the same ID. The boldfaced and/or colored dates highlight a situation where comparing the start date (STDT) with the lagging end date (LAG_ENDT) instead of LAG_MAX_ENDT could result in having BLOCKSEQ incremented erroneously.

Note that if the macro had instead compared the start date with the lagging end date (LAG_ENDT) instead of with the lagging maximum end date, the macro would have erroneously incremented BLOCKSEQ in the fifth row of data set ONE (where LAG_ENDT <= STDT – 1 but also LAG_MAX_ENDT > STDT – 1). (See dates boldfaced in Table 4.) While the fourth and fifth rows cannot by themselves combine to form a continuous block of time, they can do so when also combined with the third interval. A row-by-row implementation such as the one by Su (2007) which makes pairwise comparisons of intervals in adjacent rows may fail to detect overlaps between intervals in non-adjacent rows. (See Figure 3.)

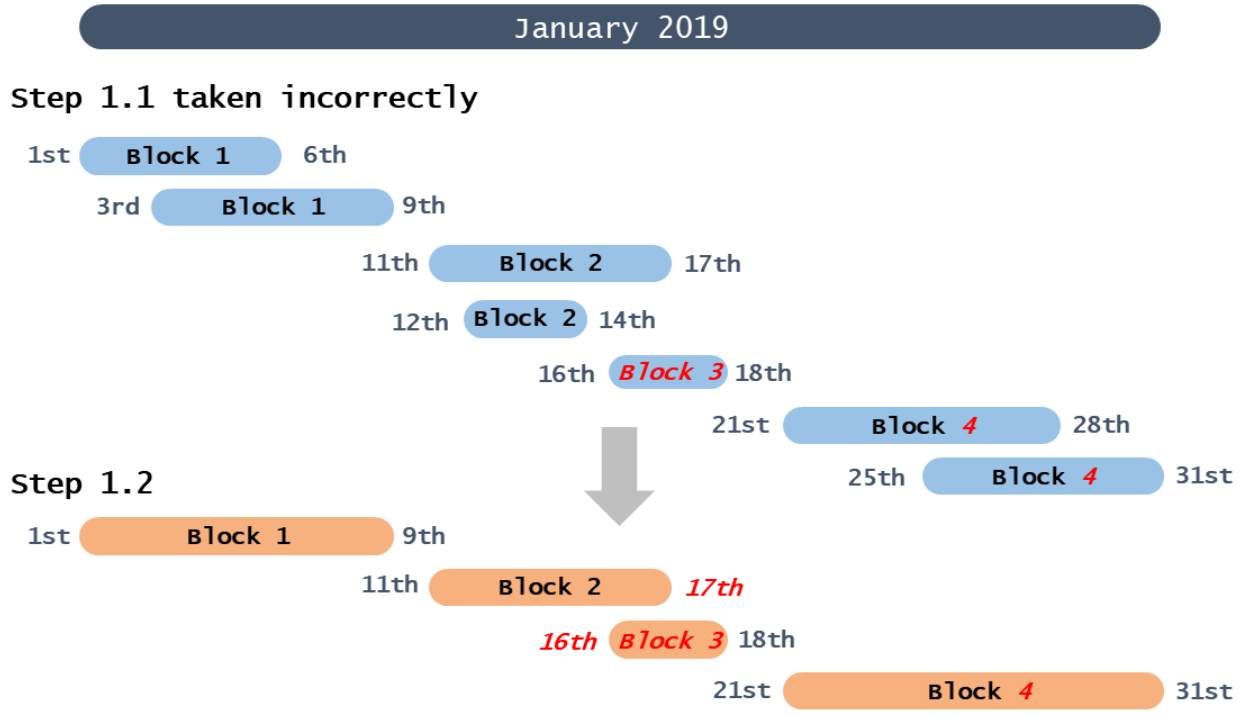

Figure 3. Comparing the start date (STDT) with only the end date of the previous row (LAG_ENDT) may result in errors (italicized in red). In this case BLOCKSEQ is erroneously incremented for the 16JAN2019-18JAN2019 interval; while it does not combine with the interval just above it (12JAN2019-14JAN2019), it still overlaps with the interval two rows above it (11JAN2019-17JAN2019). As a result, there remains an overlap on 16JAN2019-17JAN2019.

Now let's look at the SAS code "under the hood" to see how this table output was generated.

```
data _temp11;
  format usubjid blockseq lag_max_endt stdt endt; ❶
  set _temp0;
  by usubjid;

  retain max_endt; ❷

  if first.usubjid then max_endt = endt;
  else max_endt = max(max_endt, endt); ❸

  lag_max_endt = lag(max_endt); ❹

  if first.usubjid then do;
    blockseq = 1;
    lag_max_endt = .;
  end;
  else if (.z < lag_max_endt < stdt - 1) then blockseq+1; ❺

  format lag_endt max_endt lag_max_endt yymmdd10.;
run;
```

The initial format statement ❶ orders some columns so that the data is easier to skim over. The RETAIN statement ❷ and the recursive use of the MAX function ❸ together ensure that maximum end date (MAX_ENDT) is monotonically increasing for the same subject ID. MAX_ENDT is the maximum value of ENDT for the same subject (USUBJID) up to the current row. The lagging maximum end date (LAG_MAX_ENDT), is the lagging value of MAX_ENDT. ❹ BLOCKSEQ is incremented when the start date is more than one day after LAG_MAX_ENDT. BLOCKSEQ is retained implicitly by the addition of '+1' after the variable name. ❺

A word of caution: As mentioned by Su (2007), a lag function should not be placed in a conditional THEN clause. Otherwise, the lagged values may skip rows.

Step 1.2: Determine the start and end date of each combined interval.

Next, the macro determines the start and end date of each combined interval.

The output of this step is shown in Table 5. BLOCK_STDT and BLOCK_ENDT are respectively the start and end dates of a combined interval. Here we can check that each individual interval (between STDT and ENDT) is a subset of the combined interval (between BLOCK_STDT and BLOCK_ENDT).

USUBJID	BLOCKSEQ	BLOCK_STDT	BLOCK_ENDT	STDT	ENDT
101-001	1	2019-01-01	2019-01-09	2019-01-01	2019-01-06
101-001	1	2019-01-01	2019-01-09	2019-01-03	2019-01-09
101-001	2	2019-01-11	2019-01-18	2019-01-11	2019-01-17
101-001	2	2019-01-11	2019-01-18	2019-01-12	2019-01-14
101-001	2	2019-01-11	2019-01-18	2019-01-16	2019-01-18
101-001	3	2019-01-21	2019-01-31	2019-01-21	2019-01-28
101-001	3	2019-01-21	2019-01-31	2019-01-25	2019-01-31

Table 5. The data set output by step 1.2. This table has one row per interval. BLOCK_STDT and BLOCK_ENDT indicate the start and end dates of each combined interval. We can inspect the table to verify that each block (between BLOCK_STDT and BLOCK_ENDT) is a superset of all the individual intervals that compose it (between STDT and ENDT).

Here is the SAS code that outputs this table:

```
proc sql;
  create table _temp12 as
    select
      usubjid,
      blockseq,
      min(stdt) as block_stdt  format=yymmdd10., ❶
      max(endt) as block_endt  format=yymmdd10., ❷
      stdt, endt ❸
    from _temp11
    group by usubjid, blockseq ❹
    order by usubjid, blockseq, stdt, endt
  ;
quit;
```

The minimum ❶ and maximum ❷ values in PROC SQL are found for each combined interval, i.e. each group of intervals sharing the same values for USUBJID and BLOCKSEQ ❹.

STDT and ENDT ❸ are also included so that we can verify that each individual interval is a subset of the combined interval. (PROC SQL note: Since including STDT and ENDT introduces non-summary variables in the SELECT clause that are not also in the GROUP BY clause, the output table may have more than one row for each unique combination of values for the variables listed in the GROUP BY clause.)

Stage 2: Count the number of distinct days

Overview

In the second stage of the "combine-then-count" method, the macro counts the number of distinct days. Figure 4 illustrates the steps the macro takes in this stage. In Step 2.1, the macro counts the number of calendar days in each combined for interval. In Step 2.2, the macro adds up the day counts from all combined intervals. Since the combined intervals are disjoint, the calendar days they span will be distinct.

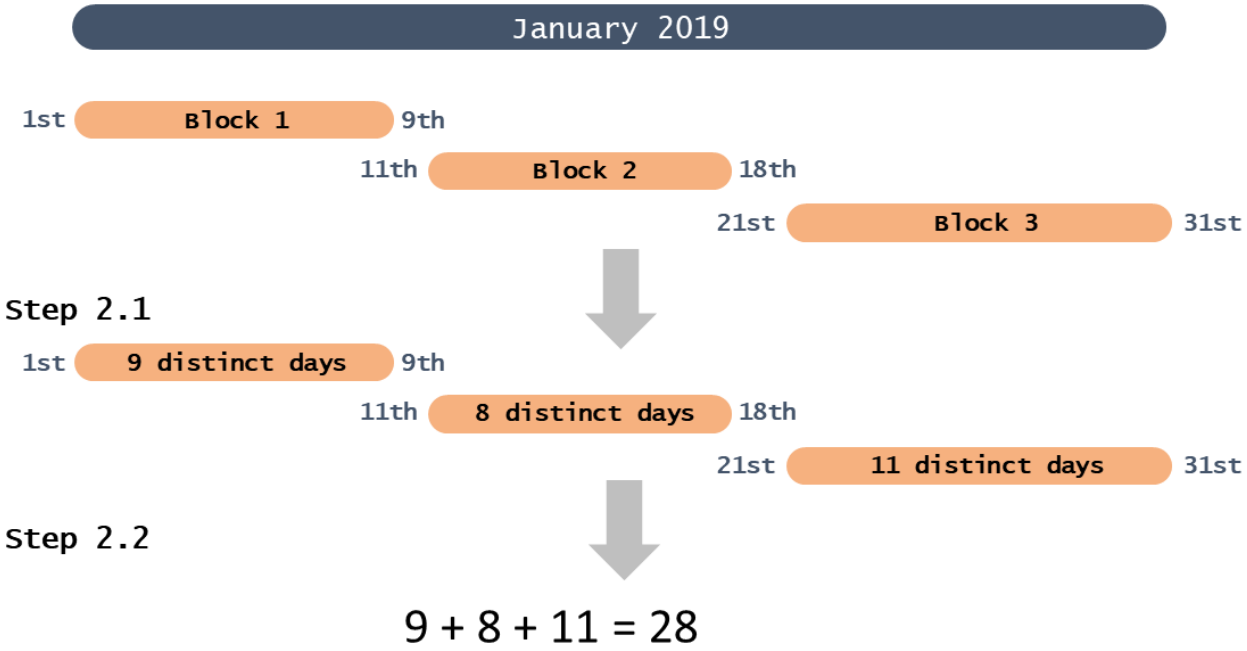

Figure 4. In the "count" stage of the "combine-then-count" method. the macro determines the number of days in each combined interval (step 2.1), then adds up the day counts from all of the combined intervals (step 2.2).

Step 2.1: Determine the number of (distinct) days in each combined interval.

In this step, the macro determines the number of distinct days in each combined interval. The output in shown in Table 6. BLOCK_DAYCOUNT is the number of distinct days in the combined interval. It is equal to BLOCK_ENDT – BLOCK_STDT + 1.

USUBJID	BLOCKSEQ	BLOCK_STDT	BLOCK_ENDT	BLOCK_DAYCOUNT
101-001	1	2019-01-01	2019-01-09	9
101-001	2	2019-01-11	2019-01-18	8
101-001	3	2019-01-21	2019-01-31	11

Table 6. The data set output by Step 2.1. This table has one row per combined interval. BLOCK_DAYCOUNT is the number of days in the combined interval, between BLOCK_STDT and BLOCK_ENDT inclusive.

Here's the SAS code that outputs this table:

```
data _temp21;
  set _temp12 (drop=stdt endt);
  by usubjid blockseq;

  block_daycount = block_endt - block_stdt + 1; ❶

  if last.blockseq; ❷
run;
```

This step calculates the number of days in each combined interval ❶, then keeps only one row per combined interval. ❷

Step 2.2: Add up the day counts from all combined intervals.

In this step, the macro adds up the day counts from all of the combined intervals. That is, it adds up all values for BLOCK_DAYCOUNT in the output from step 2.1 (Table 6).

The output from this step is shown in Table 7.

USUBJID	DAYCOUNT	BLOCKCOUNT
101-001	28	3

Table 7. The data set output by Step 2.2. This data set has one row per ID. DAYCOUNT is the number of days in all combined intervals, and BLOCKCOUNT is the number of combined intervals.

DAYCOUNT is the number of days in all combined intervals, and BLOCKCOUNT is the number of combined intervals.

Here is the SAS code:

```
proc sql;
  create table two ❶ as
    select
      usubjid, ❷
      sum(block_daycount) as dayCount, ❸
      count(unique blockseq) as blockCount ❹
    from _temp21
    group by usubjid ❺
  ;
run;
```

The name of the output dataset, TWO ❶, was specified in the macro call for this example. (See page 3.) Here the table keeps only one row for each subject ID (as USUBJID is the only non-summary variable within the SELECT clause ❷ and the only variable within the GROUP BY clause ❺). DAYCOUNT ❸ is the number of distinct days across all combined intervals for the same ID. BLOCKCOUNT ❹ is the number of combined intervals for the same ID.

CONCLUSION

The macro implementation presented in this paper follows the design principle known as "separation of concerns", keeping the task of combining of date intervals separated from the task of counting of distinct days. It treats the word "then" from the phrase "combine-then-count" as a single occurrence ("combine all intervals first, then count") rather than an iterative one ("combine two intervals at a time, then count, then repeat as necessary"). Combining the intervals yielded a set of disjoint intervals, which simplified the subsequent task of counting the distinct days within them. Preserving the information about the original intervals as they were being combined made the process of combining the intervals more transparent.

REFERENCES

Cheng, Alice M. (2006). "Duration Calculation from a Clinical Programmer's Perspective." Presented at the SAS User Group International 31 conference. http://www2.sas.com/proceedings/sugi31/048-31.pdf.

Su, Pon. (2007). "Calculating Duration via the Lagged Values of Variables." Presented at the SAS Global Forum 2007 conference. http://www2.sas.com/proceedings/forum2007/029-2007.pdf.

ACKNOWLEDGMENTS

The author thanks the following persons for their feedback: Sophia Zhang tested the macro code and detected a case overlooked by a previous version of the macro. Andrew Hansen gave helpful comments and suggestions on revising this paper. Hongxia Yan and Manyu Li had helpful questions which spurred edits to clarify assumptions about input data.

DISCLAIMER

The contents of this paper are the work of the author and do not necessarily represent the opinions, recommendations, or practices of Synteract.

CONTACT INFORMATION

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

Noory Kim
noory.kim@synteract.com

APPENDIX

The full code of the macro is below, followed by an example.

```
%macro countUniqueDays(
  inputDataset=, idVariable=, startDateVariable=, endDateVariable=,
  outputDataset=);

  /* ------------------------------------------------------------------------

     Teach us to count our days rightly, that we may obtain a wise heart.
     - Psalm 90:12

     Required parameters:
       inputDataset - name of input data set
       idVariable - identifier, e.g. subject ID
       startDateVariable - variable with numeric start date
       endDateVariable - variable with numeric end date

     Default values:
       Name of output dataset: &inputDataset._daycount
       Missing value for &endDateVariable replaced by value for
         &startDateVariable

     Names of temporary datasets: _temp0, _temp11, _temp12, _temp21

     Output data set variables:
      &idVariable
      DAYCOUNT - number of distinct days for each value of &idVariable
      BLOCKCOUNT - number of disjoint time intervals for each value of
        &idVariable

  ------------------------------------------------------------------------ */

    /* define name of outputDataset if null */
  %if &outputDataset eq  %then %let outputDataset= &inputDataset._daycount;
  %put Name of outputDataset: &outputDataset;

    /* Step 0: Sort intervals (with the same id) chronologically. */
  proc sort data=&inputDataset out=_temp0;
    by &idVariable &startDateVariable &endDateVariable;
  run;

    /* Step 1.1: Determine which intervals overlap and can thus be combined. */
  data _temp11;
    format &idVariable blockseq lag_max_endt &startDateVariable &endDateVariable;
    set _temp0;
    by &idVariable;

    retain max_endt;

    if first.&idVariable then max_endt = &endDateVariable;
    else max_endt = max(max_endt, &endDateVariable);

    lag_max_endt = lag(max_endt);
```

```
    if first.&idVariable then do;
      blockseq = 1;
      lag_max_endt = .;
    end;
    else if (.z < lag_max_endt < &startDateVariable - 1) then blockseq+1;

    format max_endt lag_max_endt yymmdd10.;
  run;

    /* Step 1.2: Determine the start and end date of each combined interval. */
  proc sql;
    create table _temp12 as
      select
        &idVariable,
        blockseq,
        min(&startDateVariable) as block_stdt  format=yymmdd10.,
        max(&endDateVariable) as block_endt  format=yymmdd10.,
        &startDateVariable,
        &endDateVariable
      from _temp11
      group by &idVariable, blockseq
      order by &idVariable, blockseq, &startDateVariable, &endDateVariable
    ;
  quit;

    /* Step 2.1: Determine the # of distinct days in each combined interval. */
  data _temp21;
    set _temp12 (drop=&startDateVariable &endDateVariable);
    by &idVariable blockseq;

    if nmiss(block_stdt, block_endt) = 0 then do;
      block_daycount = block_endt - block_stdt + 1;
    end;

    if last.blockseq;
  run;

    /* Step 2.2: Add up day counts from all combined intervals. */
  proc sql;
    create table &outputDataset as
      select
        &idVariable,
        sum(block_daycount) as dayCount,
        count(unique blockseq) as blockCount
      from _temp21
      group by &idVariable
    ;
  quit;

%mend countUniqueDays;
```

```
    /* ------------ EXAMPLE ------------ */

data one;
  input usubjid $ stdt:yymmdd10. endt:yymmdd10.;
  format stdt endt yymmdd10.;
  cards;
101-001 2019-01-01 2019-01-05
101-001 2019-01-03 2019-01-09
101-001 2019-01-11 2019-01-17
101-001 2019-01-12 2019-01-14
101-001 2019-01-16 2019-01-18
101-001 2019-01-21 2019-01-28
101-001 2019-01-25 2019-01-31
;
run;

%countUniqueDays(inputDataset=one, idVariable=usubjid,
                 startDateVariable=stdt, endDateVariable=endt,
                 outputDataset=two);
```