

A Couple of Tasty SAS® Programming Tunes

Paul M. Dorfman

CitiCorp AT&T Universal Card
Jacksonville, FL

ABSTRACT

Of all wildly different questions asked by daily participants in SAS-L jam sessions, programming problems in the strict sense of the word represent a relatively small part. An even smaller fraction belongs to interesting, difficult, or unconventional problems. While SAS programming puzzles of this nature stubbornly resist brute force, they quite often yield to a good measure of general-programming, not necessarily SAS-specific, thinking, applied before the first line of code has even been typed. When as a result, an efficient and complete SAS solution falls in place as if by magic, it feels not unlike finding a beautiful tune. One who discovers the music is rewarded by a SAS programming melody so tasty that it just must be heard outside of SAS-L. This presentation is an attempt to share a couple of such programming tunes with the SSU audience.

INTRODUCTION

In this paper, I am going to display a small miscellany of serendipitous programming problems asked on SAS-L, for which I was fortunate to offer a solution. All these problems were different in nature but shared one distinctive feature: None could be solved head-on using one of standard SAS programming techniques; or at least a standard solution, if found, would be quite inefficient due to either amount or shape of the data.

In the 'normal' SAS programming practice, the necessity to code *ab ovo*, at a very low algorithmic level, is rarely required or, should such necessity arise, an attempt is usually made to circumvent the labor by

1. Using a less efficient but standard method.
2. Buying a piece of software designed to tackle this sort of problems.
3. Asking a question on SAS-L.

The latter method works particularly well, because many SAS-L responders:

1. Are experienced, and not necessarily SAS-only, programmers.
2. Are willing to take their time to help.

3. Like programming challenges and view a difficult programming task as an interesting mental exercise.
4. Test their solutions and offer their extensions if necessary.
5. Are free.

Besides, some, very rare respondents, such as Ian Whitlock, Peter Crawford, Pete Lund, Lauren Haworth, are known to always get it right and trusted with their solutions as if they had an iron-clad guarantee coming with them.

I have always adhered to the thought that since SAS is a general programming language in its own right -- and would be even if its only component were the DATA step -- there is virtually no general programming problem that could be solved in a different software but not in SAS. By the 'general programming problem', I mean not some fancy GUI development or systems programming, but rather the implementations of schemes described in general texts on algorithms, such as Knuth[1,2], Sedgewick[3], or Standish[4].

You will see that the problems examined below pretty much fall under the cover of this loose definition. They are presented almost 'as is', with the questions as formulated by the original poster and my answers edited only for the sake of making them more suited for a paper than a personal response.

1. A KNAPSACK OF ADJUSTMENTS

This question was asked twice, almost in the same terms, by two different posters a year or so apart. Here I adhere to the thread initiated by Greg Moron gmoron@netscape.net.

Q.:

I have 50 so-called usage charges:

```
48 30 45 08 19 22 37 46 13 14
49 36 05 25 44 01 11 03 32 32
09 05 35 32 03 41 03 20 28 39
```

SERENDIPITY

40 46 47 33 03 03 08 13 50 15

06 42 20 09 32 24 16 41 21 01

I also have a large (over 3 million observations) SAS data set TARGET with 1 variable called TARGET listing 'target adjustments'. For example:

OBS	TARGET
1	53
2	278
3	115
4	75
5	264
6	278
7	156
8	153
9	15
10	20

For each target adjustment value in this file, I have to create a 'real' adjustment according to the following rules. If the target adjustment matches one of the usage charges, the charge is chosen as the real adjustment. E.g., it would be the case for TARGET=15 and 20. If a match is not found, I must try to combine 2 charges so that their sum would equal the target adjustment. For instance, for TARGET=53 the charges 50+3 would produce the required adjustment. If I can't find 2 terms like that I must try 3 terms, and so on, up to 6 terms. Generally, if a target figure can be obtained in a number of ways, a combination with fewest terms and largest possible terms must be selected. If the TARGET value can't be matched by any 1-, 2-, ..., 6-term combination, TARGET+1 value should be considered as a target adjustment, and so on. At end, I have to have a file ADJ written with TARGET and 6 variables ADJ1--ADJ6. For instance, for the first 10 observations from the file TARGET I can find 'by hand' that:

TARGET	ADJ1	ADJ2	ADJ3	ADJ4	ADJ5	ADJ6
53	50	3	.	.	.	.
278	50	49	48	47	45	39
115	50	49	16	.	.	.
75	50	25	.	.	.	.
264	50	49	48	47	46	24
278	50	49	48	47	45	39
156	50	49	48	48	.	.
153	50	49	48	48	.	.
15	15	.	.	.	.	.
20	20	.	.	.	.	.

As a matter of fact, I am not even sure I have selected the optimum required combinations for the large targets but at least, the terms total to the target adjustment. I am not so much confused about creating the combinations, rather as to how to search them efficiently in this particular case. Now -- to the response.

A.:

The task reminds the "knapsack" problem usually attacked using greedy algorithms. However, in this particular case, the number of combinations at hand does not prohibit an exact solution. The total number of 1-, 2-, ... 6-term combinations out of 50 is 18,260,635, believe it or count it. Of course, storing all of them in a kind of memory table to facilitate a comparison-type (i.e. sequential, binary, formatted, etc.) search for the needed sum each time a record comes from TARGET would be a painful experience. Moreover, it would be unclear how to make good use of equal usage charges such as 3 and 32. However, once we agree to forget about comparing TARGET to the usage charges and move from comparison-based searching to searching based on direct addressing into an array, the pieces of the puzzle fall in place as if by magic.

Let us first consider what kind of sums we are dealing with.

Rearranging the usage charges into descending order, we have:

50 49 48 47 46 46 45 44 42 41
41 40 39 37 36 35 33 32 32 32
32 30 28 25 24 22 21 20 20 19
16 15 14 13 13 11 09 09 08 08
06 05 05 03 03 03 03 01 01

Hence, the maximum sum the charges can adjust is $(50 + 49 + \dots + 48 + 47 + 46 + 46) = 286$, the minimum sum obviously being 1. Therefore, instead of creating all the combinations to search, we can only compute all possible sums from 1 to 286 that 1-, 2-, ..., 6-term combinations are able to produce. Then we will key-index an 286-bucket array by the sums corresponding to different combinations. Each sum as a search key will be used as an index into the array itself (hence key-indexing).

To account for all six possible terms, we will need a 2-dimensional array $S(1:6, 1:286)$. By creating the needed combinations from an input containing the usage charges in descending order, we can satisfy the precedence requirement naturally by not allowing the buckets already key-indexed (we will know it by the contents of the first row cell in the array S) to be overwritten by a combination having a lower priority.

Data Uc;

Input Uc @@;

Cards;

48 30 45 08 19 22 37 46 13 14

49 36 05 25 44 01 11 03 32 32

```

09 5 35 32 03 41 03 20 28 39
40 46 47 33 03 03 08 13 50 15
06 42 20 09 32 24 16 41 21 01
;
Run;

```

```
Proc Sort Data=Uc; By Descending Uc; Run;
```

```

Data _Null_;
  Do X=1 To N;
    Set Uc Nobs=N;
    If X Eq N Then Min = Uc;
    If X Le 6 Then Max ++ Uc;
  End;
  Call Symput ('Min',Compress(Put(Min,Best.)));
  Call Symput ('Max',Compress(Put(Max,Best.)));
  Call Symput ('Ucn',Compress(Put(N ,Best.)));
Run;

```

```

Data Adj (Keep=Target Adj1-Adj6);
  Array A ( 1:&Ucn) _Temporary_;
  Array S (1:6,&Min:&Max) _Temporary_;
  Array Adj (1:6 ) ;

```

```

Do X=1 To &Ucn;
  Set Uc;
  A(X) = Uc;
End;

```

```

X0 = 0;
Do X1=X0+1 To &Ucn;
  Sum = A(X1);
  If S(1,Sum) Then Continue;
  S(1,Sum) = A(X1);
End;

```

```

Do X1=X0+1 To &Ucn-1;
Do X2=X1+1 To &Ucn-0;
  Sum = A(X1)+A(X2);
  If S(1,Sum) Then Continue;
  S(1,Sum) = A(X1);
  S(2,Sum) = A(X2);
End; End;

```

```

Do X1=X0+1 To &Ucn-2;
Do X2=X1+1 To &Ucn-1;
Do X3=X2+1 To &Ucn-0;
  Sum = A(X1)+A(X2)+A(X3);
  If S(1,Sum) Then Continue;
  S(1,Sum) = A(X1);
  S(2,Sum) = A(X2);
  S(3,Sum) = A(X3);
End; End; End;

```

```
<. . Fill the gap by induction . . >
```

```

Do X1=X0+1 To &Ucn-5;
Do X2=X1+1 To &Ucn-4;
Do X3=X2+1 To &Ucn-3;
Do X4=X3+1 To &Ucn-2;

```

```

Do X5=X4+1 To &Ucn-1;
Do X6=X5+1 To &Ucn-0;
  Sum = A(X1)+A(X2)+A(X3)+A(X4)+A(X5)+A(X6);
  If S(1,Sum) Then Continue;
  S(1,Sum) = A(X1);
  S(2,Sum) = A(X2);
  S(3,Sum) = A(X3);
  S(4,Sum) = A(X4);
  S(5,Sum) = A(X5);
  S(6,Sum) = A(X6);
End; End; End; End; End; End;

```

```

Do Until (Eof);
  Set Target End=Eof;
  If Not (&Min Le Target Le &Max)
    Then Continue;
  Do While (Not S(1,Target));
    Target ++ 1;
  End;
  Do X=1 To 6;
    Adj(X) = S(X,Target);
  End;
  Output;
End;
Run;

```

The Data _Null_ step was used to find the minimum and maximum ranges pertaining to the usage charges and create macro variables for sizing up arrays in the subsequent step.

In the code above, the combination loops are kept unrolled for the sake of initial clarity. However, now it is now clear by induction how to write a macro capable of assembling this code, the last step can be made much more concise. For example, the needed macro could be coded this way:

```

%Macro Cmb(N);
  %Local Z;
  %Do Z=1 %To &N;
    Do X&Z=X%Eval(&Z-1)+1 To &Ucn-(&N-&Z);
  %End;
  Sum = A(X1)
  %Do Z=2 %To &N; +A(X&Z) %End;;
  If S(1,Sum) Then Continue;
  %Do Z=1 %To &N; S(&Z,Sum) = A(X&Z); %End;
  %Do Z=1 %To &N; End; %End;
%Mend Cmb;

```

Now the macro can be used to shorten the program dramatically:

```

Data Adj;
  Array A ( 1:&Ucn) _Temporary_;
  Array S (1:6,&Min:&Max) _Temporary_;
  Array Adj (1:6 ) ;
  Do X=1 To &Ucn;
    Set Uc;
    A (X) = Uc;
  End;

```

SERENDIPITY

```
X0 = 0;

%Cmb (1) %Cmb (2) %Cmb (3)
%Cmb (4) %Cmb (5) %Cmb (6)

Do Until (Eof);
  Set Target End=Eof;
  If Not (&Min Le Target Le &Max) Then
    Continue;
  Do While (Not S(1,Target));
    Target ++ 1;
  End;
  Do X=1 To 6;
    Adj(X) = S(X,Target);
  End;
  Output;
End;
Run;
```

Computing the combinatorial sums and preparing the key-indexed table is the longest and most slowly executed part of the program. When the last combination macro has executed, the table is ready for searching. The TARGET file is then read in an explicit DO loop, and each target value coming from its observations is searched via a single array reference.

How long does it take the computer to solve this, at the first glance, CPU and I/O intensive problem? With 1,000,000 observations in TARGET, going over all 18,260,635 combinations, using them to key-index the sums, and computing the adjustments takes about a minute, in all, on a 233MHz P-II machine running 6.12 under Windows NT.

2. KEY-LINKING

This type of problem arises frequently in practical applications, especially in business situations where a customer is identified by a key (credit card number, telephone number) that is likely to change due to a number of circumstances. A customer, for instance, could lose a credit card in which case a new number is issued, while the old number remains in the database as a secondary key. Numerous changes of this kind produce a chain of keys that, in the absence of a unified shadow key, sooner or later needs to be traced. At the end of 1998, a problem of that sort was raised on SAS-L for the first time by Ludwig Boltzmann, and a number of people, notably Ian Whitlock, Karsten Self, and I offered solutions that basically differed in terms of methods different responders had employed to search the columns with 'new' and 'old' keys.

Here I am presenting the most recent version of this problem formulated and posted by Max Zwingli

maxzwingli@mail.nu and my response, in which I took an opportunity to do an exercise in structured SAS programming comparing LINK and macro approaches, and, frankly, to throw in a couple of arguments for hashing as a searching technique.

Q.:

I've got a SAS file with 2 variables: OLD and NEW. They represent old and new phone numbers once the customer's phone has changed. When this happens the old-new obs is simply inserted somewhere in the file, no ordering or indexing of any kind, and the numbers have nothing to do with the time the record was inserted. Here's a snapshot of how the data might look like:

OLD	NEW
8888888888	9999999999
2222222222	3333333333
5555555555	6666666666
7777777777	8888888888
0000011111	0000022222
3333333333	4444444444
9999999999	0000000000
1111111111	2222222222

A human eye will easily see that first-old to last-new chains are:

1. 5555555555-6666666666
2. 7777777777-8888888888-9999999999-0000000000
3. 0000011111-0000022222
4. 1111111111-2222222222-3333333333-4444444444

The file having upwards of 800000 observations, I need to devise a programmatic means of doing this. Actually with the sample like above the output should look like this:

OLD	NEW	NLINKS
5555555555	6666666666	2
7777777777	0000000000	4
0000011111	0000022222	2
1111111111	4444444444	4

Where NLINKS represents the number of links in each "chain". So far my efforts have resulted in about fifteen steps of sorts and merges taking quite some time to run. Could anyone suggest a more elegant and/or efficient approach with fewer passes through the input?

A.:

The principal algorithm for solving the problem is quite simple:

1. Allocate 3 hash tables:
 - Containing OLD phones
 - Containing NEW phones
 - Containing NEW phones *parallel* to the first table.
2. Read the file record by record and load the hash tables.
3. Read a record from the file again. Search for OLD phone in the NEW hash table. If the search is unsuccessful, the OLD phone is the beginning of a chain. Otherwise go to step 3.
4. Take the corresponding NEW phone and search it in the OLD table. If a match is found, take the corresponding NEW phone from the hash table 3 and search the OLD table again. Otherwise it is the end of the chain, so stop and output the endpoints, then go to step 3.

Now we see that the problem boils down to repeated hash searches of the same kind, so it makes sense to concoct a unified procedure and apply it (with a degree of flexibility) throughout the program. First, let us try to make use of the Macro Language to create an %HSEARCH() routine that can chameleon itself depending upon the table and key it is searching. I assume that the input data set is called A.

```
%Let H = 1000003; * Prime Number => Nobs*2;
```

```
%Macro Hsearch (Table=, Key=);
  %If      %Ucase(&Table) = %Ucase(Old)
    %Then %Let Table = 1;
  %Else %If %Ucase(&Table) = %Ucase(New)
    %Then %Let Table = 2;
  Found = 0;
  Do J=1+Mod(&Key,&H) By 1
    Until (H(&Table,J)=. Or Found);
    If J = &H Then J = 1;
    If H(&Table,J) = &Key Then Found = 1;
  End;
%Mend Hsearch;
```

```
*H-dimensions: 1=Old, 2=New, 3=(Old||New) ;
```

```
Data Oldnew (Keep=Old New Nlinks);
  Array H (3,&H) _Temporary_;
  Do Until (E1);
    Set A End=E1;
    %Hsearch (Table=Old, Key=Old);
    H(1,J) = Old;
    H(3,J) = New;
    %Hsearch (Table=New, Key=New);
    H(2,J) = New;
  End;
  Do Until (E2);
    Set A End=E2;
    %Hsearch (Table=New, Key=Old);
    If Found Then Continue;
```

```
Do Nlinks=2 By 1 Until (Not Found);
  %Hsearch (Table=Old, Key=New);
  If Found Then New = H(3,J);
End;
Output;
End;
Run;
```

A good number of people dislike the Macro Language whenever a more conventional means of structured programming can be employed. I do not necessarily share that viewpoint given certain SAS limitations, but let us give the LINK subroutine a fair shot as well:

```
*H-dimensions: 1=Old, 2=New, 3=(Old||New) ;
```

```
Data Oldnew (Keep=Old New Nlinks);
  Array H (3,&H) _Temporary_;
  Do Until (E1);
    Set A End=E1;
    Table = 1; Key = Old; Link Hsearch;
    H(1,J) = Old;
    H(3,J) = New;
    Table = 2; Key = New; Link Hsearch;
    H(2,J) = New;
  End;
  Do Until (E2);
    Set A End=E2;
    Table = 2; Key = Old; Link Hsearch;
    If Found Then Continue;
    Table = 1;
    Do Nlinks=2 By 1 Until (Not Found);
      Key = New; Link Hsearch;
      If Found Then New = H(3,J);
    End;
    Output;
  End;
  Stop;
Hsearch:
  Found = 0;
  Do J=1+Mod(Key,&H) By 1
    Until (H(Table,J) = . Or Found);
    If J = &H Then J = 1;
    If H(Table,J) = Key Then Found = 1;
  End;
Run;
```

In both cases, the problem is solved in a single step, albeit with two passes through the file. Above, hash table size is assumed about twice the number of the keys to search, in order to obtain the fastest searching speed with the simplest collision resolution policy -- the linear probing.

3. TOPOLOGICAL SORT

SERENDIPITY

This very intriguing problem was posted by Alex Martchenko amartchenko@netscape.net, who called it 'Job Scheduling'. This problem, too, lends itself to a kind of structured DATA step programming, but here it is more of a necessity, otherwise the algorithm is difficult to follow given the code.

Q.:

I have a bunch of 'jobs', and all I know about them is which must be executed before which. Suppose the info is recorded in a SAS data set 'pairs'. Every observation in PAIRS tells that some J_BEf must be performed before J_AFT.

```
DATA PAIRS;
INPUT J_GRP J_BEf $8. J_AFT $8.;
LINES;
1 III BBB
1 CCC GGG
1 GGG EEE
1 EEE HHH
1 HHH FFF
1 DDD FFF
1 AAA CCC
1 GGG DDD
1 III EEE
1 BBB HHH
3 555 888
3 555 777
3 666 999
3 444 666
3 000 111
3 222 444
3 111 222
3 111 333
3 000 999
3 333 666
3 444 777
3 222 555
;
RUN;
```

I need to identify distinct jobs within each group (1 and 3 above) and output them to a variable JOB_SEQ so that no job listed in J_BEf for that group follows J_AFT. I don't care about the exact output sequence as long as this rule is not violated. For example the output done by hand may look like shown below (in the 3-group I on purpose selected the jobs pairs to make the output look sorted like 000, 111, 222, 333...):

Obs	J_GRP	JOB_SEQ
1	1	III
2	1	AAA
3	1	BBB
4	1	CCC
5	1	GGG
6	1	EEE
7	1	DDD
8	1	HHH
9	1	FFF
10	3	000

11	3	111
12	3	222
13	3	333
14	3	444
15	3	555
16	3	666
17	3	777
18	3	888
19	3	999

In the example I used 3 byte character values for simplicity but in actuality J_BEf and J_AFT have full 8 bytes. Also even though the file is pretty big (> 21 m obs) each group is limited to no more than 500 records. Questions: 1) Is there a SAS procedure that can accomplish the task; 2) if not has someone an idea how to program it in base SAS. Frankly, I have none. First I decided it'd be easy to do with a couple of sorts and datasteps but the more I think of it, the more hazy it seems.

A.:

What we have got here is the simplest scheduling problem with a topological sort to be done within each by-group. Assuming that you have no cyclic references, it is fairly easy. At least the basic idea of the algorithm lies on the surface: Let us first find all the jobs having *no* predecessors. Apparently, we can place them in the output right away. Then if we remove them from the input, thus erasing their 'before-after' relationships with the remaining items, some other jobs will emerge with no predecessors. Repeating the process until the input has been exhausted, we will finally have output a linear list of items where no successor is listed before any of its predecessors.

Telling the computer to do the same is, however, a more intricate matter. First, it is convenient (and apparently more efficient) to (1) determine the number of predecessors of each job beforehand. Having obtained such a frequency, we can then safely (2) output the items with zero frequencies, since they have no predecessors. Then (3) for each item like that, we also want to look at each of its successors and decrease its predecessor count by a unity. As a result of this operation, some other items will end up having zero counts. Now we can go back to (3), repeating the process until all counts have been zeroed out. To facilitate it, we need to associate a list of successors with each job item. Since there are only 500 items maximum per group, it can be done using a 2-dimensional 500x500 array instead of (more complex) collection of coalesced link lists. Of course, most of the memory space occupied by the array nodes will be wasted, but since it is only 2 Mb, one can afford to sacrifice it for simplicity.

The scheme described above might sound simple, but it is inefficient to scan the entire (modified) input for zero counts each time the process iterates. This can be avoided by maintaining an output queue. After the *initial* scan for zeroes the queue can be initialized by the items having zero predecessor counts. Then we eject the job sitting in the front of the queue, place it in the output, and decrease the predecessor counts of all its successors. If any of them have gone down to zero, the corresponding item is inserted in the rear of the queue. The new front queue item is ejected again, and the loop repeats until the queue is empty and there are no successors to deal with, at which point the goal is apparently reached.

Let us consider the first group:

```
1 III BBB
1 CCC GGG
1 GGG EEE
1 EEE HHH
1 HHH FFF
1 DDD FFF
1 AAA CCC
1 GGG DDD
1 III EEE
```

and do the algorithm by hand - it will make the way of writing a program strikingly clear. First, let us enumerate all the distinct jobs in the group:

```
III -> 1, BBB -> 2, CCC -> 3, GGG -> 4, EEE->5,
HHH -> 6, FFF -> 7, DDD -> 8, AAA -> 9
```

Second, let us read the input and populate the following table using the enumerating numbers instead of the actual job names:

	1	2	3	4	5	6	7	8	9	10	11	12	...	500
Pre_Cnt	0	1	1	1	2	1	2	1	0	.	.	.	.	.
Suc_Cnt	2	0	1	2	1	1	0	1	1	.	.	.	.	.
Suc1	2	.	4	5	6	7	.	7	3	.	.	.	.	.
Suc2	5	.	.	8	.	.	.	.	.	.	.	.	.	.
Suc3	.	.	.	.	.	.	.	.	.	.	.	.	.	.
....	.	.	.	.	.	.	.	.	.	.	.	.	.	.
Suc500	.	.	.	.	.	.	.	.	.	.	.	.	.	.

According to the plan above, since the items 1 and 9 have zero predecessor counts, we grab them and insert in the rear of the queue one by one, so the initialized queue (rear to front as left to right) now looks like Q[9,1].

Item 1 is in the front of the queue, so we take it and move to the output: OUT[1]. Item 1 has two successors: 2 and 5. Decreasing the predecessor count if item 2 yields 0, so 2 goes in the rear of the queue, and thus now the queue is Q[2,9]. Decreasing PRE_CNT of item 5 yields 1, so we leave

it alone -- for now. Since 9 has moved to the front of the queue, we take it and move to the output: OUT[1,9]. But item 9 has item 3 as its successor, so we should decrement PRE_CNT of item 3 by 1. It yields 0, so 3 goes in the queue: Q[3,2]. At this point, the table has acquired the following form (only non-missing rows and columns shown):

	1	2	3	4	5	6	7	8	9
Pre_Cnt	0	0	0	1	1	1	2	1	0
Suc_Cnt	2	0	1	2	1	1	0	1	1
Suc1	2	.	4	5	6	7	.	7	3
Suc2	5	.	.	8	.	.	.	.	.

Next item in front of the queue is 2, and so it goes to the output: OUT[1,9,2]. Item 2 has no successors, so we turn our attention back to the queue, which at the moment is Q[3]. Item 3 goes to the output: OUT[1,9,2,3], and it has item 4 as its sole successor. Subtracting 1 from PRE_CNT(4) knocks it down to zero, so 4 is inserted in the queue: Q[4]. There are no more successors to process for item 4, so we go back to the queue and eject item 4 into the output: OUT[1,9,2,3,4]. Its successors are 5 and 8. Decreasing PRE_CNT of both makes them ripe for the queue, and that is where they go: Q[8,5]. Front item 5 goes to the output: OUT[1,9,2,3,4,5]. Its successor 6 has PRE_CNT(6)=1, so decreasing it by 1 results in 0, and 6 goes in the queue: Q[6,8]. Item 8 is in front, so it is output: OUT[1,9,2,3,4,5,8]. It then causes PRE_CNT(7) of its only successor 7 to go down by 1, and the queue, now Q[6], spits out its only entry: OUT[1,9,2,3,4,5,8,6]. The only successor of 6, item 7, has thereby its PRE_CNT gone down to 0, and so it goes in the rear of the queue: Q[7]. Item 7 goes to the output list: OUT[1,9,2,3,4,5,8,6,7]. This terminates the process, since there are no more successors to get involved with, and the queue is empty: Q[.]. All we have to do now is replace the numbers with their respective job names:

```
JOB_SEQ [III AAA BBB CCC GGG EEE DDD HHH FFF].
```

Obviously, the scheme is very easy to program if the jobs are contiguously enumerated, for because the table is indexed by item numbers, any item in question is in effect located immediately by key-indexed search. Otherwise we would have to conduct a search of a different type - and nothing comes close to key-indexing in speed and simplicity. Therefore, it makes sense to spend some time enumerating unique items within each by-group. It can be done on the fly, and the most natural medium to do it is a hash table. We simply allocate a hash table JN_HSH of a prime size about 2*500 and try inserting the next job item. If it is the first time in, the item gets the next

SERENDIPITY

serial number recorded in a parallel table J_To_N, and the inverse relationship is recorded in a table N_To_J (apparently having to house maximum 500 nodes). If the item is duplicate, we simply use the number already sitting in this hash location of J_To_N.

Organizing a queue is even easier. (Note that here, SAS's ready-to-go method of organizing a queue, LAG, cannot go anywhere: We need a queue that would facilitate ejection of the item from the front asynchronous with insertion in the rear - while LAG does both simultaneously.) An array of size 500 will do the job if we use 2 pointers, REAR and FRONT, initially positioned at 0 and 1. To insert, we increment REAR by a unity and stick an item in QUEUE(REAR). To eject, we output QUEUE(FRONT) and increment FRONT, effectively moving the next array node to the front of the queue. FRONT = REAR means the queue having become empty.

Neither the QUEUE, nor PRE_CNT array has to be cleaned up before the next by-group: The ex-contents of the queue matter not when the FRONT/REAR pointers are initialized; and at the end of the process, all predecessor counts are zeroed out. However, both the hash table and the successor table have to be cleaned up properly.

The DATA step program below solves the scheduling problem within given limitations (no more than 500 distinct jobs per group). Writing the code, I have significantly deviated from my own non-structured, parsimonious practice in hope that it could make the intent of the instructions more transparent. You be the judge if I have failed or not. If you prefer a more concise style, you can, for instance, pack arrays Pre_Cnt, Suc_Cnt, and Queue in 0, -1, and -2 rows of Suc_Lst, shorten notation, and so on.

```
Data Pairs;
  Input J_Grp J_Bef: $8. J_Aft: $8.;
Cards;
1 III BBB
1 CCC GGG
1 GGG EEE
1 EEE HHH
1 HHH FFF
1 DDD FFF
1 AAA CCC
1 GGG DDD
1 III EEE
1 BBB HHH
3 555 888
3 555 777
3 666 999
3 444 666
3 000 111
3 222 444
3 111 222
```

```
3 111 333
3 000 999
3 333 666
3 444 777
3 222 555
2 LLL MMM
2 LLL YYY
2 TTT XXX
2 VVV TTT
2 SSS VVV
2 UUU SSS
2 UUU WWW
2 XXX QQQ
2 WWW TTT
2 VVV YYY
2 SSS LLL
;
```

Run;

```
%Let M = 500; *Max Distinct Jobs Per Group;
%Let H = 1009; *Hash Size For Enumeration ;
```

Data Schedule (Keep=J_Grp Job_Seq);

```
Array Suc_Lst (1:&M, 1:&M) _Temporary_;
Array Pre_Cnt (1:&M) _Temporary_;
Array Suc_Cnt (1:&M) _Temporary_;
Array Queue (1:&M) _Temporary_;
Array N_To_J (1:&M) $8 _Temporary_;
Array J_To_N (1:&H) _Temporary_;
Array JN_Hsh (1:&H) $8 _Temporary_;
```

Link Init;

```
Do Until (Last.J_Grp);
  Set Pairs;
  By J_Grp Notsorted;
```

```
J = J_Bef; Link Enum; Pre_X = J_To_N (X);
J = J_Aft; Link Enum; Suc_X = J_To_N (X);
```

```
Pre_Cnt (Suc_X) ++ 1;
Suc_Cnt (Pre_X) ++ 1;
Suc_Lst (Pre_X, Suc_Cnt(Pre_X)) = Suc_X;
End;
```

```
Do Q = 1 To N;
  Link Q_Insert;
End;
```

```
Do Front = 1 By +1 Until ( Front = Rear );
  Link Q_Eject;
  Do X=1 By +1 While ( Suc_Lst(Q_Front,X) );
    Q = Suc_Lst (Q_Front,X);
    Pre_Cnt (Q) +- 1;
    Link Q_Insert;
  End;
End;
```

```
Link Clean_Up;
Return;
```

Init:

```

Do X = 1 To &H;
  JN_Hsh (X) = ' ';
  J_To_N (X) = 0 ;
End;
N      = 0;
Rear   = 0;
Return;

Enum:
  Do X = Mod(Input(J,Pib6.),&H) + 1 By +1
    Until ( JN_Hsh(X) = J );
  If H > &H Then H = 1;
  If Not J_To_N (X) Then Do;
    N ++ 1      ;
    J_To_N (X) = N;
    N_To_J (N) = J;
    JN_Hsh (X) = J;
  End;
End;
Return;

Q_Insert:
  If Not Pre_Cnt (Q) Then Do;
    Rear ++ 1;
    Queue (Rear) = Q;
  End;
Return;

Q_Eject:
  Q_Front = Queue (Front);
  Job_Seq = N_To_J (Q_Front);
  Output;
Return;

Clean_Up:
  Do X = 1 To N;
    Suc_Cnt (X) = 0;
    Do Q = 1 To N;
      Suc_Lst (X, Q) = 0;
    End;
  End;
Return;
Run;

```

The topological sort program can be modified in order to detect vicious cycles (that is, intransitive relations of the type B->W W->Z Z->B). On a more philosophical note, the problem of topological sort is similar to that of delineating a directed acyclic graph.

CONCLUSION

SAS problems and, by consequence, SAS programs are not created equal. Some, even a majority, eagerly yield to a relatively few ready-to-go SAS coding techniques. Others, probably a minority, are more stubborn and need actual programming to tackle them. On the other hand, their solutions are quite

useful to those asking for help and satisfactory to those finding solutions. Playing good music is a great gift. But finding a new, beautiful melody is unlike anything else.

SAS is a registered trademark or trademark of SAS Institute, Inc. in the USA and other countries. ® indicates USA registration.

REFERENCES

1. D. E.Knuth, *The Art of Computer Programming*, 2.
2. D. E.Knuth, *The Art of Computer Programming*, 3.
3. R. Sedgewick, *Algorithms in C*, Parts 1-4.
4. T. A. Standish. *Data Structures, Algorithms and Software Principles in C*.

ACKNOWLEDGMENTS

My hat goes off to Ian Whitlock for providing his friendly support for this unconventional kind of SAS programming on SAS-L. I would also like to thank Ian for the invitation to present these SAS programming tunes in SSU Serendipity Section.

And this paper would have never been written if it had not been for my wife Doris, her infinite patience and non-intrusive care.

AUTHOR CONTACT INFORMATION

Paul M. Dorfman,
SAS Programmer

10023 Belle Rive Blvd. 817,
Jacksonville, FL 32256

(904) 564-1931 (h)
(904) 905-5428 (o)

sashole@bellsouth.net
paul_dorfman@hotmail.com