Programming the HP 33s

Martin Cohen

1. Introduction

The HP 33s is an odd calculator. It is based on the HP 32sII, which has the following characteristics:

4-level stack
384 program steps
26 program labels (A-Z)
34 variables (A-Z, i, statistics)
some complex operations (complex = 2 reals)
conditional statement execution (if condition then do instruction)
subroutine call

In particular, the limited program space, labels, and variables are reasonably balanced restrictions.

The HP 33s is based on the 32sII. It contains the same features (though recoded to run on a different processor by Kenpo) with the following additions:

32,768 bytes for programs and data (!!!)
an algebraic mode where expressions can be used
some additional instructions (cube root, integer divide and remainder)
a few bugs
a hard-to-see decimal point
an unusual (but usable) keyboard

2. Programming style for the 33s

The greatly expanded memory makes for an interesting coding style (at least for me). Essentially, there is no limit to the size of programs that can be entered. However, the limitations on labels and variables are exactly the same as on the 32sII. This causes the following to be the first commandment of 33s programming:

Use As Few Labels As Possible

When code of the form “if condition then do stuff” occurs, instead of writing it as “if not condition goto label a do stuff label a”, if stuff takes more than one instruction, I try to write it as “if condition do inst1 if condition do inst2 ...”, so that no labels are used.
For example, in the code shown later for solving a 3x3 linear system of equations, the determinant of the system is stored in W and values are stored in X, Y, and Z that need to be divided by W if W is non-zero. If labels were not so scarce,

I would write
rcl W
x=0?
goto A
sto÷ X
sto÷ Y
sto÷ Z
lbl A

However, to save a label, I wrote it as
rcl W
x≠0?
sto÷ X
x≠0?
sto÷ Y
x≠0?
sto÷ Z

By repeating the test, I saved a label.

A similar idea can be used to save labels when as expression (not statement) of the form "if condition then value1 else value2" occurs. The idea is to compute both value1 and value2 and then, depending on condition, put the desired result in a known stack location. Explicitly, the code could be (if the condition is "v1=v2")

compute value1
compute value2
get v2
get v1 (value2 in Z-reg)
x=y? (if v1=v2, put value1 in Z-reg)
R↓
R↓ (put desired value in X-reg)
R↓

I realize that "R↑" could be used and save one instruction, but remember: On the 33s, instructions don't matter as much because there is plenty of memory; labels and variables do.

3. How I got started

I got my 32sII when I read that they were to be discontinued. I didn't want to miss out, as I had done on the last batch of HP 42s. I had previously written a number of programs for the 48GX, all in user RPL. The 32sII was interesting to program, but limiting because of the small program space. Whenever I wanted to put in a new program, I had to remove anything already there. This took a lot of the fun out.

When the 33s became available, I ordered one. I figured that, with the essentially infinite program memory, I could have a lot more fun with it. I was right.

The first major program I wrote was one to solve a 3x3 linear system of equations. I read the example program that did this in the manual, and I was upset that the program there destroyed the original

coefficients. In a calculator with so little data storage as the 33s, original data should be preserved whenever possible.

The equations were to be

A*X+B*Y+C*Z=D
E*X+F*Y+G*Z=H
I*X+J*Y+K*Z=L

I remembered Cramer's rule, which solves linear system of equations as rations of determinants. The most complicated subroutine would be one that computed the determinant

|A B C|
|E F G|
|I J K|

The sample code had a routine that did this, but I (naturally) wrote my own.

The next most complicated routine swapped the column (D, H, L) with any of the columns (A, E, I), (B, F, J) or (C, G, K). The main routine would then do this:

compute determinant; store in W
swap (D, H, L) with (A, E, I)
compute determinant; store in X
swap (D, H, L) with (A, E, I) (restoring the original coefficients)
swap (D, H, L) with (B, F, J)
compute determinant; store in Y
swap (D, H, L) with (B, F, J)
swap (D, H, L) with (C, G, K)
compute determinant; store in Z
swap (D, H, L) with (C, G, K)
if W≠0, divide X, Y, and Z by W

Voila – the equations are solved and the original coefficients are unchanged!

I also wrote routines to input the coefficients and check the results (a good thing, too!). I took advantage of some idiosyncrasies of the 33s conditionals to enable the input routine to, in addition to just entering values, either clear the coefficients to zero or store random values there.

This code (and most of the code discussed here) was posted to hpmuseum.org and comp.sys.hp48 and listed in Appendix A.

4. Something new for the 33s – bit fiddling

The 33s could display results in hex, octal, and binary, but it could only do the standard operations it already had. It could not do logical operations (and, or, xor, ...), and I needed them. So, I decided to write them.

My first conclusion was that the only reasonable way to do this was to examine each corresponding bit of the two operands and compute the result. I first thought that each operation (and, or, ...) would need some individualized code to do its stuff, but then I thought of having a truth table with one bit for each of the four possible inputs (00, 01, 10, and 11). This worked nicely. This code was also posted. Two of the new 33s instructions (integer divide and remainder) proved quite useful.

It then occurred to me that the routine could be sped up by processing two bits at a time. The truth table needed would be 32 bits long (2 bits times 16 possible inputs), and so would fit nicely into a 33s integer (which is 36 bits long). This was also coded. As a sidelight, I wrote a routine that took as input a 4-bit (1-bit input) truth table and produced a 32-bit (2-bit input) truth table. I used this to check my hand-generated truth tables, finding some mistakes.

My final version, and the one presented here (listed in Appendix B), is a parameterized version that takes as input the truth table, the number of bits processed at each step (1 or 2), and the number of bits in each value to process (usually 16 or 32). The result is the bit-fiddled combination of the inputs.

An interesting foible of the 33s occurred while entering the code. The truth tables, especially the 32-bit ones, are best entered in hex mode. However, the code, which required using the y^x operator, had to be entered in decimal mode since that operator (and the whole top row of keys) becomes a hex digit in hex mode.

5. Sorting

On July 15, 2004, Steve Adam posted to comp.sys.hp48 33s code to sort variables A through Z. I liked the code, which used an insertion sort. It used i as the index in the insertion loop, and the value at (33) (one of the stat variables) to save the count of items sorted so far.

I set myself the task of writing a sort routine that worked as well and had the following improvements:

1. Any initial set of variables (e.g., A through K) could be sorted;
2. No other variable besides i would be used to hold loop information;
3. No other storage would be used.

Appendix C shows my code that satisfies all these requirements. The main techniques used are:

1. The number of items to be sorted is kept in the stack, and the code is carefully constructed to preserve this value;

2. The integer part of i is used in the insertion loop, and the fractional part of i is used to hold the number of items sorted so far.

6. Linear least squares

Note: This exposition assumes a modest knowledge of calculus.

The linear least squares (LLS) problem is that of best approximating a set of (x, y) data in the least squares sense by a linear combination of functions of x. More equationistically, we are given a set of n data points (x_i, y_i) and a set of m functions of a single variable, f_k, and we want to find the coefficients a_k such that $F(x_i) = \Sigma_k a_k f_k(x_i)$ best approximates y_i for i from 1 to n.

As a notational convenience, I write Σ_k for the sum for k from 1 to m (sum over the functions) and Σ_i for the sum for i from 1 to n (sum over the data points).
The expression to be minimized is
$D = \Sigma_i(y_i - \Sigma_k a_k f_k(x_i))^2$.

This is the sum of the squares of the differences. A weighted sum could also be used with only a small increase in complexity, but that is left as an exercise for the reader.
D is minimized by taking its partial derivative with respect to each of the a_k, setting that to zero, and solving the resulting linear system of equations for the a_k.

Let Dj be the partial derivative of D with respect to a_j. Then

$Dj = \Sigma_i(y_i - \Sigma_k a_k f_k(x_i))(2f_j(x_i))$
$= \Sigma_i 2y_i f_j(x_i) - \Sigma_i(2f_j(x_i))(\Sigma_k a_k f_k(x_i))$
$= 2\Sigma_i y_i f_j(x_i) - 2\Sigma_k a_k \Sigma_i f_j(x_i)\ f_k(x_i)$ (reversing the order of summation)
Setting Dj to zero (and dropping the factor of 2), the a_k must be the solutions to the linear set of equations $\Sigma_k a_k \Sigma_i f_j(x_i)\ f_k(x_i) = \Sigma_i y_i f_j(x_i)$ for j from 1 to m.

If we write $A_{j,k} = \Sigma_i f_j(x_i)\ f_k(x_i)$ and $B_j = \Sigma_i y_i f_j(x_i)$, this assumes the form

$\Sigma_k a_k A_{j,k} = B_j$ for j from 1 to m.

As examples, we consider the cases when m=1 and m=2.
When m=1, we are approximating data by a single function. A typical case would be approximating data by a line through the origin. Then $f_1(x) = x$, and we want an approximation of the form y = a*x. The single equation that we get is (writing a for a_1) $a\ \Sigma_i x_i^2 = \Sigma_i x_i y_i$.

When m=2, we are approximating data by a linear combination of two functions. The canonical example of this is the good old linear least squares fit, where $f_1(x) = 1$ (constant) and $f_2(x) = x$, so we get an approximation of the form a_1+a_2x.

The resulting equations are (writing a for a_1 and b for a_2)

$a\ \Sigma_i 1*1 + b\Sigma_i 1*x_i = \Sigma_i y_i*1$ (j = 1)
$a\ \Sigma_i x_i*1 + b\Sigma_i x_i*x_i = \Sigma_i y_i*x_i$ (j = 2)

or, in the traditional form,
$a * n + b\Sigma_i x_i = \Sigma_i y_i$ (j = 1)
$a\ \Sigma_i x_i + b\Sigma_i x_i^2 = \Sigma_i x_i y_i$ (j = 2)

The goal for the 33s is to write code that will allow any f_k to be used in the fit. This requires that (1) we have a method for specifying the f_k, (2) we can accumulate the sums that occur in the linear equations for the a_k, and (3) we can solve the linear systems.

Doing (3) is easy – we just happen to have code for solving linear 3x3 and 2x2 systems.
To do (1), we allocate one of our precious labels (I use F here, since that is what I use in my 33s) to a routine that, when passed a value of x in the x-reg, returns $f_1(x)$, $f_2(x)$, and $f_3(x)$ in, respectively, the x-reg, y-reg, and z-reg. It is ok if fewer than 3 functions are computed by this routine, since, when we solve the linear system, we will solve an m by m system where m is the number of functions actually computed by the routine.

As an example, to fit a+b*x to data, the code for F would be
LBL F (x is in the x-reg – we want 1 and x)
1 (1 in x-reg, x in y-reg, don't care about the z-reg)
RTN

Another example: To fit a*x+b*sin(x)+c*cos(x) to data, the code for F could be

LBL F (x is in the x-reg – we want x, sin(x), and cos(x))
ENTER
ENTER (x in x, y, z-regs)
SIN
x<>y
COS (cos(x) in x-reg, sin(x) in y-reg, x in z-reg)
RTN

To do (2), we have to get around the problem of the 33s having only 26 locations for data, and 12 of them (A-L) are used for storing the coefficients of the linear equations to be solved. My solution is to use the largest available hunk of 33s storage – program space! Each data point (x, y) is passed to a routine (I used Q since not many labels were available to me) that calls routine F with x and accumulates the sums for a 3x3 system. My code in Q stores x and y in variables X and Y so they can be used (if needed) in F.

After all the sums have been accumulated, the user chooses the equation solver depending on the number of functions being fitted – label S for 3 functions, label T for 2 functions, and manually computing D÷A for 1 function. This is why the routine (label T) for solving a 2x2 system uses a subset of the 3x3 system coefficients.

So, the code for fitting data looks like this:

```
0 XEQ I     ( initialize coefficients to zero
yval1       ( do this for each data point
```

```
xval1       ( xval, yval in x-reg, y-reg
XEQ Q       ( accumulate the point
...         ( repeat for each data point
RTN         ( done
```

When all the point have been processed, the user (that's you) does XEQ S for 3 functions, XEQ T for 2 functions, or RCL D RCL A ÷ for 1 function.
The code for this is in Appendix D.

As a side note, once the a_k have been computed, D itself can be readily gotten. To show this,

$D = \Sigma_i(y_i - \Sigma_k a_k f_k(x_i))^2$
$= \Sigma_i(y_i^2 - 2y_i\Sigma_k a_k f_k(x_i) + (\Sigma_k a_k f_k(x_i))^2)$
$= \Sigma_i y_i^2 - 2\Sigma_i y_i\Sigma_k a_k f_k(x_i) + \Sigma_i(\Sigma_k a_k f_k(x_i))^2)$
$= S_2 - 2\Sigma_k a_k\Sigma_i y_i f_k(x_i) + \Sigma_i(\Sigma_k a_k f_k(x_i))(\Sigma_j a_j f_j(x_i))$ where $S_2 = \Sigma_i y_i^2$
$= S_2 - 2\Sigma_k a_k B_k + \Sigma_k a_k\Sigma_j a_j\Sigma_i f_k(x_i)_j f_j(x_i)$ where the B_k are from the equations for a_k
$= S_2 - 2\Sigma_k a_k B_k + \Sigma_k a_k\Sigma_j a_j A_{j,k}$ where the $A_{j,k}$ are from the equations for a_k

But, from the equations, $\Sigma_k a_k A_{j,k} = B_j$ for j from 1 to m. Swapping the roles of j and k, the expression for D becomes

$D = S_2 - 2\Sigma_k a_k B_k + \Sigma_k a_k B_k = S_2 - \Sigma_k a_k B_k$.

So, if we modify routine Q to also accumulate the sum of y_i^2 (and remember to initialize it at the start), we can easily get D. For example, in the 2 function case, after getting the results (which are stored in X and Y), if $\Sigma_i y_i^2$ is stored in V, and since B_1 and B_2 are in variables D and H, the value of D is
V – D*X – H*Y. The first line should be changed from "`0 XEQ I`" to "`0 sto V XEQ I`".

Appendix A. Solving linear systems of equations

These are my routines for solving 2x2 and 3x3 linear equations. They use Cramer's rule (ratio of determinants) to solve the equations. The coefficients are not changed when the equations are solved. This is useful on the 33s, where so few values can be stored.

The routines are ("det" means "determinant"):

S: Solve 3x3 linear system
(A,B,C;E,F,G;I,J,K).(X,Y,Z)=(D,H,L)
det in W
Result in W (det), X, Y, Z;
Display is det in X, then X, Y, Z in y-reg, z-reg, t-reg, respectively
R↓ if display ≠ 0 to see X, Y, Z

T: Solve 2x2 linear system
A*X+B*Y=D, E*X+F*Y=H
Det in W; displayed when computed
Result in X and Y and regs
Note these are a subset of the 3x3 systems,
instead of the expected
A*X+B*Y=C, D*X+E*Y=F

U: Get det(A,B,C;E,F,G;I,J,K)
V: Swap column (D;H;L) with col (X-reg)
I: Input A..L (uses label J)
Asks for inputs if X>0; clears if X=0; sets to random if X<0.

```
      Comments indicated by "//".

S: Solve 3x3 linear system
      (A,B,C;E,F,G;I,J,K).(X,Y,Z)=(D,H,L)
      det in W
      Result in W (det), X, Y, Z;
      Display is det in X-reg, then X, Y, Z
      R↓ if display ≠ 0 to see X, Y, Z

Or, in matrix form,

(A B C) (X)   (D)
(E F G).(Y) = (H)
(I J K) (Z)   (L)

LBL S
XEQ U STO W // GET DET
1 XEQ V XEQ U STO X // GET DET SWAPPING COL 1 WITH (D,H,L)
1 XEQ V // RESTORE MATRIX
2 XEQ V XEQ U STO Y // SAME FOR COL 2
2 XEQ V
3 XEQ V XEQ U STO Z // SAME FOR COL 3
3 XEQ V
RCL W x≠0? STO÷ X // IF DET NOT ZERO, SET X
x≠0? STO÷ Y // SAME FOR Y
x≠0? STO÷ Z // SAME FOR Z
RCL Z RCL Y RCL X RCL W // GET RESULTS
RTN // DONE

T: Solve 2x2 linear system
      A*X+B*Y=D
      E*X+F*Y=H
      Det in W; displayed when computed
      Result in X and Y and regs
      Note these are a subset of the 3x3 systems,
      instead of the expected
      A*X+B*Y=C, D*X+E*Y=F

LBL T
A*F-B*E
STO W // SAVE THE DET
VIEW W // SHOW IT
D*F-B*H // DET FOR X
STO X // STORE IT
```

```
A*H-D*E // DET FOR Y
STO Y // STORE IT
RCL W // GET DET
x≠0? STO÷ X // IF NON-ZERO, GET TRUE X
x≠0? STO÷ Y // IF NON-ZERO, GET TRUE Y
RCL Y RCL X RCL W // GET RESULTS
RTN // DONE

U: Get det(A,B,C; E,F,G; I,J,K)
The matrix is:
| A B C |
| E F G |
| I J K |

LBL U
RCL F RCL* K // F*K
RCL G RCL* J // G*J
- RCL* A // A*(F*K-G*J)
RCL B RCL* K // B*K
RCL C RCL*J // C*J
- RCL* E - // THAT - E*(B*K-C*J)
RCL B RCL* G // B*G
RCL C RCL* F // C*F
- RCL* I + // THAT + I*(B*G-C*F)
RTN // THAT'S THE DET

V: Swap column (D;H;L) with col (X-reg)

LBL V
STO i // STORE COLUMN INDEX (1=(A,E,I), 2=(B,F,G), 3=C,G,E))
RCL D x<>(i) STO D // SWAP D WITH FIRST ROW
4 STO+ i // POINT TO NEXT ROW
RCL H x<>(i) STO H // SWAP H WITH SECOND ROW
4 STO+ i // POINT TO NEXT ROW
RCL L x<>(i) STO L // SWAP L WITH THIRD ROW
RTN // THAT'S IT

I: Input A..L (uses label J)
      Inputs if X>0; clears if X=0; sets to random if X<0.

LBL I
STO X // SAVE INPUT
1.012 STO i // SET LOOP CONTROL
LBL J
RCL X // GET CONTROLLER
x>0? INPUT (i) // IF > 0, ASK FOR VALUE
x<0? RANDOM // IF < 0, GET A RANDOM VALUE
STO (i) // STORE WHAT YOU GOT
ISG i GTO J // LOOP
RTN // DONE AFTER 12

K: Check all equations, stacking the differences
      Note that this uses algebraic mode

LBL K
I*X+J*Y+K*Z-L // check third equation – diff ends up in the z-reg
E*X+F*Y+G*Z-H // check second equation
A*X+B*Y+C*Z-D // check first equation
RTN // THAT'S THE DIFFERENCES
```

Appendix B. Bit-fiddling

Bit-fiddling on the HP 33s

Routines:

H: return the number of bits in each value to process (called size, usually 16 or 32) in the x-reg and the number of bits at a time to process (called width, 1 or 2) in the y-reg.

D: given the truth table for the operation in the x-reg, value1 in the y-reg, and value2 in the z-reg, return operation(value1, value2) in the x-reg. use label E and variables T (truth table), U (value1), V (value2), W (2*width – 2 or 4), S (2^(size-width)), and R (the result).

A: returns x-reg and y-reg.

O: returns x-reg or y-reg.

X: returns x-reg exclusive-or y-reg.

L: given a 4-bit truth table, return the corresponding 32-bit truth table. use label M. this code assumes that routine H specifies 1-bit width.

Code:

```
LBL A // and (view in hex mode)
8 gto D // width 1, 4-bit table (1000)
E4A04400 gto D // width 2, 32-bit table)

LBL O // or (view in hex mode)
E gto D // width 1, 4-bit table (1110)
FFEEF5E4 gto D // width 2, 32-bit table

LBL X // exclusive or (view in hex mode)
6 gto D // width 1, 4-bit table (0110)
1B4EB1E4 gto D // width 2, 32-bit table

LBL N // not
// originally, I did a truth table for this.
// when I posted this code to comp.sys.hp48, someone pointed out
//    that not(x) = -(x+1)
//    (which I knew but didn't apply –
//    "when all you have is a hammer, everything looks like a nail")
enter enter 1 + +/-
rtn

LBL H
2     // width (1 or 2)
32    // size (usually 16 or 32)
RTN

LBL D
```

```
sto T // store truth table
R↓ abs sto U // store value1, positively
R↓ abs sto V // store value2
xeq H // get size and width
x<>y sto W // W = width
2 sto* W // W now = 2*width
R↓ - // size-width
2 x<>y y^x // 2^(size-width)
sto S // S = initial shift factor
0 sto R // initialize result

LBL E // the fiddling loop
rcl V // get value2
rcl S int/ // divide by shift factor
rcl W rmdr // leave only the wanted bits
rcl U // same for value1
rcl S int/
rcl W rmdr
rcl* W // scale value1 bits up
+ // combine with value2 bits
rcl W x<>y y^x // get (2*width)^combined bits for truth table lookup
rcl T // the truth table
x<>y int/ // shift right by lookup amount
rcl W rmdr // get just the bits wanted
rcl* S // shift left to same position as input bits
sto+ R // store into result
rcl S rcl W int/ // get next bit location to the right
sto S // save that
x>0? gto E // if not done, go again
rcl V rcl U rcl R // done – load inputs and output
rtn

LBL L
sto Y // store 4-bit truth table
0 sto Z // 32-bit table built up in Z
16 sto i // count 16 possibilities

LBL M // the loop
rcl i pse // show where at
ip // in case have a fractional part
1 - // make 0 to 15 (4 bits)
enter // copy it
enter // again
4 int/ // high 2 bits
x<>y 4 rmdr // low 2 bits
rcl Y xeq D // assumes H returns width 1, size at least 2
4 rmdr // get last 2 bits
4 rcl i 1 – y^x // get 4^(0 to 15)
* // scale result up
sto+ Z // store into result
rcl Z pse // show me
dse i gto M // do until done
rcl Y rcl Z // show input and output
rtn
```

Appendix C. Sorting

These routines sort A..(any letter) into increasing order. The variable i is used to hold both a current pointer (in its integer part, IP) and the number of values stored so far (in its fractional part, FP, after dividing by 100). Using (i) makes use of the fact that the 33s ignores the FP of i.

There are two routines. The core routine (label Y, also using label Z), adds a new value to the currently sorted list. Routine label R sorts the first n values (in A to whatever), where n is the value in the X register. i must be set to zero before calling R (to save a label).

Routine Y is carefully designed to preserve the X and Y registers passed to it and leave them as they were when it returns. This allows routine R to keep the number of items to be sorted in the registers so it does not need to be stored.

Here are the routines.

```
Routine Y.

Inputs: X register: value to be inserted into sorted list.
        i: IP=current pointer; FP=number in list/100

Outputs: The value is placed in the list and the list size is incremented.
         The X and Y registers are as when called.

Lbl Y rcl i IP x=0? gto z // if before list start, done
R↓ // value back to X
rcl (i) x<y? gto Z // if (i)<value, done
1 sto+ i R↓ sto (i) R↓ // store (i) into (i+1), preserving X, Y
2 sto-i // change incremented i to i-1
R↓ R↓ // leave the value in X (dropping 2 and (i))
gto Y // do again
Lbl Z // value in Y reg, want to store in i+1
R↓ 1.01 sto+ i // value in X reg, bump i by 1.01
R↓ // remove 1.01
sto (i) // store val
rcl i FP 101 x sto i // recover i from FP, put in IP and FP
R↓ // leave value in X reg
rtn // done

Routine R.

Inputs: You must set i to zero (this foolishness saves a label).
        Put number of items (A to whatever) in the X register.
        (I.e., to sort A through I, place 9 there).

At the end, the values should be sorted.

Lbl R rcl i IP pse x>=y? rtn // done if pointer >= number wanted
R↓ // restore number to X
1 sto+ i rcl (i) // get next value
x<>y sto- i R↓ // restore i and restore stack
XEQ Y // insert the value
R↓ gto R // remove value from stack (so size to sort in X) and repeat
```

Appendix D. Linear least squares

The coefficients for the equations to be solved are shown here with the indicies indicating the sums stored there – (j,k) means $\Sigma_i f_j(x_i)\ f_k(x_i)$ and (j) means $\Sigma_i y_i f_j(x_i)$.

```
A(1,1) B(1,2) C(1,3) D(1)
E(2,1) F(2,2) G(2,3) H(2)
I(3,1) J(3,2) K(3,3) L(3)
```

```
lbl Q       ( accumulate sums for fitting 3 functions
sto X x<>y sto Y  ( save x and y
ENTER x² sto+ V R↓ ( use this if accumulating sum(y²)
x<>y xeq F  ( get f1, f2, f3 in regs
sto U R↓ sto V R↓ sto W ( save f1 in U, f2 in V, f3 in W
(     accumulate y*fj in D, H, L
rcl Y rcl* U sto+ D
rcl Y rcl* V sto+ H
rcl Y rcl* W sto+ L
( accumulate fj*fk in left-hand side
rcl U x² sto+ A          ( f1*f1
rcl U rcl* V sto+ B sto+ E    ( f1*f2 = f2*f1
rcl U rcl* W sto+ C sto+ I    ( f1*f3 = f3*f1
rcl V x² sto+ F          ( f2*f2
rcl V rcl* W sto+ G sto+ J    ( f2*f3 = f3*f2
rcl W x² sto+ K          ( f3*f3
RTN
```