Sample 1. Unstable summation.

1. Do the sum in double precision.

```
    k =   0    s =     1.750000000000000E+01
    k =   5    s =    -4.955444729786392E+08
    k =  10    s =     6.361137619195046E+11
    k =  15    s =    -6.829259024353563E+12
    k =  20    s =     3.022687119371480E+12
    k =  25    s =    -1.224830662857075E+11
    k =  30    s =     7.315904098222299E+08
    k =  35    s =    -8.834682915557073E+05
    k =  40    s =     2.701921995744262E+02
    k =  45    s =     2.070038693322906E-02
    k =  50    s =     4.546179247215157E-02
    k =  55    s =     4.546101674944338E-02
    k =  60    s =     4.546101675867138E-02
    k =  62    s =     4.546101675867133E-02
```

2. Use FM with increasing precision.

```
    Setting precision to j digits via call fm_set(j) will actually set the
    equivalent number of decimal significant digits slightly higher than j.
    For example, if the base used internally in FM is 10**7, then asking for
    20 digits with call fm_set(20) gives at least 29 significant digits.
    call fm_set(30) gives at least 36 significant digits.
    call fm_set(40) gives at least 50 significant digits.
    call fm_set(50) gives at least 57 significant digits.

    20 digits,  73 terms gave   s_fm =   .04399094217962562151
             This calculation lost about  18 digits.
    30 digits,  78 terms gave   s_fm =   .043990942179625639969699565464
             This calculation lost about  18 digits.
    40 digits,  88 terms gave   s_fm =   .0439909421796256399696989706597424719113
             This calculation lost about  18 digits.
    50 digits,  93 terms gave   s_fm =   .04399094217962563996969897065974247192700503984511
             This calculation lost about  18 digits.
```

3. Use FM with increasing precision in base 2.

```
    Using  53 bits,  62 terms gave   s_fm =  .045461016758671 33
              This calculation lost about  14 decimal digits.
    Using  73 bits,  67 terms gave   s_fm =  .04399094194167266143888
              This calculation lost about  14 decimal digits.
    Using  93 bits,  72 terms gave   s_fm =  .043990942179628 13948568386511
              This calculation lost about  15 decimal digits.
    Using 113 bits,  77 terms gave   s_fm =  .043990942179625639968630235187619 58
              This calculation lost about  14 decimal digits.
```

4. Use FM with different rounding modes in base 2.

```
    Using  53 bits, rounding left         , 62 terms gave   s_fm =   .02897151338476893
    Using  53 bits, rounding symmetrically, 62 terms gave   s_fm =   .04546101675867133
    Using  53 bits, rounding right        , 62 terms gave   s_fm =   .05194200656637992
    These agree to about  0 decimal digits.
```

```
     Using 113 bits, rounding left         , 77 terms gave   s_fm =   .04399094217962563
     Using 113 bits, rounding symmetrically, 77 terms gave   s_fm =   .04399094217962564
     Using 113 bits, rounding right        , 77 terms gave   s_fm =   .04399094217962564
     These agree to about 18 decimal digits.

 5.  Use FM with interval arithmetic in base 2.

     Using  53 bits,  62 terms gave  s_fm_interval =  [  -.1021766113637601 ,   .1945740695617867 ]
     The two endpoints agree to about  0 decimal digits.

     Using 113 bits,  77 terms gave  s_fm_interval =  [   .0439909421796257 ,   .0439909421796257 ]
     The two endpoints agree to about 18 decimal digits.


Summary:

For this calculation all four methods for measuring the degree of
instability worked well.  When done with double precision carrying
16 significant digits and using the default symmetric rounding,
only 1 digit remained correct at the end of the sum.

Interval arithmetic is probably the strongest of these checks, and
using FM arithmetic with 30 digits and a large base (method 2) is
the fastest of these methods for getting the sum correct to full
double precision accuracy.

Comparing the last value of s in method 1 with the first value of
s_fm in method 3 should show whether this compiler carries extra
digits while evaluating expressions like x / ( (k+1) * fact**2 ).
If the two values are the same, no extra digits are carried in d.p.




 Sample 2.  Unstable recurrence.

     1.  Use non-interval FM arithmetic with 30 digits.

         After  5 terms with 30 digit accuracy, the result is    4.91084749908279
         After 10 terms with 30 digit accuracy, the result is    4.99277028806207
         After 15 terms with 30 digit accuracy, the result is    4.99943593714684
         After 20 terms with 30 digit accuracy, the result is    4.99995612709047
         After 25 terms with 30 digit accuracy, the result is    5.00009039073834
         After 30 terms with 30 digit accuracy, the result is   77.16163142845431
         After 35 terms with 30 digit accuracy, the result is   99.99999060423245
         After 40 terms with 30 digit accuracy, the result is   99.99999999999706
         After 45 terms with 30 digit accuracy, the result is  100.00000000000000
         After 50 terms with 30 digit accuracy, the result is  100.00000000000000
         After 55 terms with 30 digit accuracy, the result is  100.00000000000000
         After 60 terms with 30 digit accuracy, the result is  100.00000000000000

     2.  Use interval arithmetic with 30 digit accuracy.

         After  1 terms the result is [  4.4705882352941176 ,  4.4705882352941176 ]
         After  2 terms the result is [  4.6447368421052632 ,  4.6447368421052632 ]
         After  3 terms the result is [  4.7705382436260623 ,  4.7705382436260623 ]
         After  4 terms the result is [  4.8557007125890736 ,  4.8557007125890736 ]
```

```
After  5 terms the result is [  4.9108474990827932 ,  4.9108474990827932 ]
After  6 terms the result is [  4.9455374041239167 ,  4.9455374041239167 ]
After  7 terms the result is [  4.9669625817627006 ,  4.9669625817627006 ]
After  8 terms the result is [  4.9800457013556312 ,  4.9800457013556312 ]
After  9 terms the result is [  4.9879794484783923 ,  4.9879794484783923 ]
After 10 terms the result is [  4.9927702880620681 ,  4.9927702880620681 ]
After 11 terms the result is [  4.9956558915066340 ,  4.9956558915066340 ]
After 12 terms the result is [  4.9973912683813441 ,  4.9973912683813441 ]
After 13 terms the result is [  4.9984339439448169 ,  4.9984339439448169 ]
After 14 terms the result is [  4.9990600719708939 ,  4.9990600719708939 ]
After 15 terms the result is [  4.9994359371468391 ,  4.9994359371468392 ]
After 16 terms the result is [  4.9996615241037670 ,  4.9996615241037680 ]
After 17 terms the result is [  4.9997969007134058 ,  4.9997969007134286 ]
After 18 terms the result is [  4.9998781354776765 ,  4.9998781354781580 ]
After 19 terms the result is [  4.9999268794992229 ,  4.9999268795094152 ]
After 20 terms the result is [  4.9999561269476688 ,  4.9999561271634113 ]
After 21 terms the result is [  4.9999736736100334 ,  4.9999736781766820 ]
After 22 terms the result is [  4.9999841549419453 ,  4.9999842516043860 ]
After 23 terms the result is [  4.9999894553132943 ,  4.9999915013665818 ]
After 24 terms the result is [  4.9999717607369067 ,  5.0000150694764154 ]
After 25 terms the result is [  4.9995202522450967 ,  5.0004369664045127 ]
After 26 terms the result is [  4.9899353973829161 ,  5.0093395647887771 ]
After 27 terms the result is [  4.7869972899223478 ,  5.1977831184897810 ]
After 28 terms the result is [   .3000582860996429 ,  9.0357119352120193 ]
After 29 terms the result is [  -.1646377251287M+4 , 52.4812789961537590 ]
After 30 terms the result is [          - OVERFLOW ,          + OVERFLOW ]
```

Summary:

The general solution of this recurrence has a term that causes
the values to converge to 100, but for these initial conditions
4, 4.25, the specific solution has a coefficient of zero on that
term, so this sequence converges to 5 mathematically.

When rounding errors occur in later x-values, they introduce a
very small but nonzero amount of the term that causes convergence
to 100, and that term grows rapidly and soon swamps the rest of
the solution.

Interval arithmetic tracks the growing uncertainty in the x-values,
and when an interval gets big enough to include zero, dividing by
that interval is undefined and the result is
[-overflow , +overflow].

Interval arithmetic works better than a sequence of increasing
precision FM results here, since comparing FM results at 30, 40, 50
digits gives 100 each time.

Sample 3. Unstable sum.

```
For x = -25.00 The sum gave  [ 1.3887943864963056M-11 , 1.3887943864964986M-11 ]
                     correct =  1.3887943864964021M-11

For x = -30.00 The sum gave  [ 9.3576229646761189M-14 , 9.3576229730042186M-14 ]
```

```
                        correct =  9.3576229688401746M-14

     For x = -35.00 The sum gave  [ 6.3046409205537791M-16 , 6.3055928170842012M-16 ]
                        correct =  6.3051167601469894M-16


Summary:

As the input x becomes more negative, the instability increases.
This is shown as the left and right endpoints of the interval
result agree to fewer digits.

The mathematically correct value of the sum lies within the
interval in each case.



All results were ok.
```