Boldly Going - Climbing Project Euler

© 2020 Valentín Albillo

Welcome to a new article in my *"Boldly Going"* series, this time dealing with ***Project Euler***, a fantastic project which attempts (and succeeds, in spades !) at providing a huge supply of challenging math+programming problems which are immensely fun to try and solve, learning loads of new concepts while sharpening your skills in the process. In this article I tell my story with ***PE*** and how I dealt with it using the vintage 1984 ***HP-71B*** handheld computer/calculator (emulated & physical), thus enormously upping the ante.

Introduction

While enjoying my summer vacations back in 2011, yours truly serendipitously discovered ***Project Euler***, an Internet-based project which, in their own words *(my highlighting)*, ***PE*** ...

> *"... is a series of challenging mathematical/computer programming problems that will require more than just mathematical insights to solve. Although* ***mathematics*** *will help you arrive at elegant and efficient methods, the use of a computer and* ***programming skills*** *will be required to solve most problems. The motivation for starting PE [...] is to provide a platform for the inquiring mind* ***to delve into unfamiliar areas*** *and* ***learn new concepts*** *in a* ***fun*** *and* recreational *context"*

In order to check it, I tried my hand at ***PE #101*** using my virtual ***HP-71B*** (J-F Garnier's *Emu71/DOS*), quickly succeeded and got instantly hooked. As my *out-of-home* first half of the vacations had already elapsed and I was just beginning the *at-home* second half, I decided to try and solve as many problems as I could during the 15 remaining days, after which no more time could be allocated to it because of work, etc. and I'd be done with it.

The Rules

Before starting for good, I set me up some *carved-in-stone* ***rules***:

1) I would use use just a virtual and/or physical *HP-71B* for all the problems, to see how many I could solve with such a truly ancient system (namely *Emu71/DOS* running on the 16-bit subsystem of 32-bit *Windows XP* on a pretty obsolete *2000-era 2.4 Ghz* single-core *CPU* with 512 Mb of *RAM*).

2) I would write all code (utilities included) in the *HP-71B BASIC* language, augmented with the *Math, HP-IL* and *JPC ROMs*, the string-handling *LEX* files *STRINGLX* and *REPLEX*, and nothing else.

3) I absolutely would ***not*** search the Internet for solutions or hints or other people's code or whatever. The only allowable use of the Internet would be for reference (*Wikipedia*, *MathWorld*, *Wolfram*, etc.) or occasionally *OEIS* for identifying sequences but nothing else. No hints accepted, no spoilers of any kind.

4) Any new concepts would first be learned, then applied. After solving the problem, I'd immediately go solve the next one, ***no*** peeking at the *solution thread* to see how other people did (see rationale below).

5) *PE* problems are intended to be solvable in ~1' but this assumes modern hardware and high-level programming languages. As I'd be using *ancient* hardware/software I equalized by allowing for more time.

PE states that about *1 min.* should be adequate for most problems but I didn't delude myself about what my setup could physically achieve, so considering that it ran at least *100-1,000× slower* than using modern languages on multi-core *CPU*s I cut myself some slack in that regard and thus considered that anything around *10'-20'* for the virtual *71B* or *2-4d* for the *280×* slower physical one should be considered *success*.

Indeed, using a handheld calculator (even if emulated) is already handicap enough to further compound the situation by requesting physically unrealizable times, so some fair scaling was definitely in order. Not to say that I didn't strive for faster times if at all possible, e.g.: my solution for ***PE #162*** ran in just a few seconds while for the much harder ***PE #214*** my running time was under *20'*. Success.

As for the *solution threads* to which I had access after having solved a problem, I never visited them or posted my *HP-71B* solutions there, mainly for two reasons: first, I wasn't interested as this was a completely *private* endeavour so I just entered my computed result in *PE*, got the green *Ok*, and that was it, next problem please.

To be fair, I visited the solution threads just *once* right after solving my very first *PE* problem ***(PE #101)*** in order to see how the threads went, and after a cursory glance I decided that I didn't want to see any of it lest I'd spoil me both the fun and the learning, and that's my second reason.

I'll explain: it wasn't unfrequent that I did manage to solve one of the problems yet I wasn't fully satisfied with my approach. Later, I usually *revisited* the problem and came up with a much better approach, all on my own. That additional satisfaction and the self-learning which came with improving it would have been *completely ruined* if I had simply looked at the *solution threads* straight away. No second attempt would have been possible.

Also, I was on *PE* strictly for the ***fun***, not to *"learn"* from others. Any and all learning would be *self-taught*, by working hard through a problem till I solved it satisfactorily. That's how I learned the most, through sheer effort, and not only new techniques but also the fine art of finding worthy *resources* (books, papers, *PhD* thesis). Simply looking at other people's solutions, which requires no effort whatsoever, absolutely pales in comparison and is but the easy, lazy approach which, as I said before, might spoil both the fun and the learning.

The Equipment

You can't go mountain-climbing without having ready the adequate equipment beforehand (of course I didn't think about all of what follows at once, some of it came out of experience). For *PE*-climbing I'll recommend having at hand the following tools, utilities (all specific for the *HP-71B*) and references as a start:

Number Theory:	Factorization, primality testing, modular powers, primes/*Fibonacci* generators/lists, *GCD*, *LCM*, sums-/num-/lists-of-divisors, *Euler's* totient, *Moebius*, *Moebius* inversion.
Assorted:	Linear recurrences, direct formula for the *N*-th term, diophantine equations, inequalities Generation of combinations, variations and permutations, permutation-checking. Root finding, matrix operations including linear systems and determinants, basic geometry.
References:	*Wikipedia*, *MathWorld*, *Wolfram Alpha*, *OEIS*, anything that helps and doesn't spoil the fun.

And last but certainly not least, a certain level of *math proficiency* can do wonders for *PE* problems. Consider ***PE #276***, for instance, a truly wonderful problem which can be stated in one line and everyone can understand exactly what is asked, yet a *straight brute-force* attack is doomed to failure from the start.

Should you attempt such primitive approach you'll quickly find out that the problem is $O(N^3)$, thus completely unmanageable for the 10^6 limit asked. After some thinking and a little math reasoning it's possible to reduce it to $O(N^2)$, which is a *million* times faster, yet it would still take a number of months or years to arrive at a solution.

Then, if your math foundations are solid and sound, you'll eventually find a way to reduce the complexity to *O(N)*, which is a *trillion* times faster than the brute-force $O(N^3)$ and this finally delivers a correct solution in reasonable times (mine was *19'*). Even better times are still possible but that would be going for the *A+* and I was in a hurry ...

In short, improving your math skills is both *a prerequisite for* and *a consequence of* ***PE*** problem-solving.

The Techniques

Again, I didn't immediately stumble upon these useful techniques all at once but as I started from ***PE#001*** onwards (save for ***PE#101***) I eventually developed them and began to use them for every problem. Some are:

1) Before doing any math or programming, first create a text file (say, *PE104.txt*) for *every PE* problem you're going to tackle. Use it to record your notes, links to useful references, the listings of the various versions you create, from the very first, crude attempt to the final *"production"* program, as well as the results obtained when running each and your comments on them, the code for any short utilities you create to try new approaches, *anything* and *everything*. It will prove invaluable to keep a record of your efforts and as a reference when solving similar problems or even revisiting and improving each one.

2) Now, try and duplicate the sample values given for each problem (say, the result for *$N=10^4$*, where the result for *$N=10^{10}$* is later asked) to check that your initial *no-matter-how-crude-and-inefficient* algorithm works Ok, then use it to gather additional data (say, for *$N=10^k$* for *k=1,2,3,4,5,6*). Also, if you *can't* duplicate the sample values, give this problem a miss for the time being and go try some other.

   Once you have enough data (4-6 terms are usually sufficient) use some *71B* utility (like my *LINREC*, see ***References***, or write your own) to detect *patterns*, e.g.: if the data satisfy some *recurrence relation*, which can be extremely useful to greatly speed up your program and bring additional insight. This can be done as well for intermediate sequences your program finds midway. You can also try *OEIS*, it might identify the sequences and offer new terms *(but be extra-careful not to spoil anything)*, which will be useful to refine or reorient your search for patterns and eventually implement a successful algorithm.

3) You can speed your program by *pre-computing* things, using a file containing a set of pre-calculated data (think long sequences that take a while to compute). Once created, you can then retrieve data as fast as you can read them from the file which, *RAM* permitting, could possibly hold *1,000*'s of elements.

4) Don't be afraid to use *recursion*, it can be a very powerful asset either to completely solve a problem or to gather enough data for pattern recognition, and even a slow, inefficient recursive procedure will be suitable for that. It's frequently the case that recursion is a *natural* for a problem, avoiding clumsy, non-recursive procedures, and though *HP-71B*'s *BASIC* language does support recursive subprograms, at times it might be convenient to implement it using arrays instead, substituting recursive calls involving already computed elements by simpler, faster array retrieving.

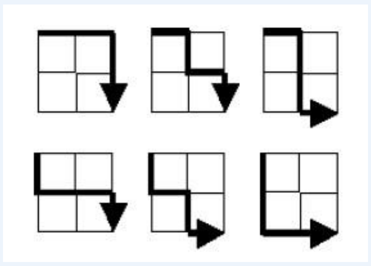
Boldly going ...

Now, at long last, I'll give here my *original (2011) HP-71B* commented solutions for the following ***7*** choice ***PE*** problems. This is less than *1%* of all currently existing *PE* problems as of 2020, so I'll grant you permission to *"cheat"* and examine them at leisure as long as you promise not to cheat at all for the remaining *99%:*

Project Euler problem ***#015*** *— Lattice paths*
Project Euler problem ***#017*** *— Number letter counts*
Project Euler problem ***#040*** *— Champernowne's constant*
Project Euler problem ***#077*** *— Prime summations*
Project Euler problem ***#093*** *— Arithmetic expressions*
Project Euler problem ***#094*** *— Almost equilateral triangles*
Project Euler problem ***#104*** *— Pandigital Fibonacci ends*

Project Euler #015: ***Lattice paths***

Starting in the top left corner of a 2×2 grid, and only being able to move to the right and down, there are exactly 6 routes to the bottom right corner.



How many such routes are there through a 20×20 grid?

My *139-byte* solution for the ***HP-71B*** is:

```
10  DESTROY ALL @ STD @ OPTION BASE 1 @ L=20 @ DIM R(L,L)
20  FOR I=1 TO L @ R(I,1)=I+1 @ R(1,I)=RES @ NEXT I
30  FOR I=2 TO L @ FOR J=2 TO L @ N=1 @ FOR K=1 TO I @ N=N+R(K,J-1) @ NEXT K
40  R(I,J)=N @ NEXT J @ DISP I;R(I,I) @ NEXT I
```

```
[RUN]  →   2   6                { 6 routes through a 2×2 grid }
           3   20
           4   70
           5   252              { 252 routes through a 5×5 grid }
           6   924
           7   3432
           8   12870
           9   48620
          10   184756           { 184,756 routes through a 10×10 grid }
          11   705432
          12   2704156
          13   10400600
          14   40116600
          15   155117520        { 155,117,520 routes through a 15×15 grid }
          16   601080390
          17   2333606220
          18   9075135300
          19   35345263800
          20   137846528820     { 137,846,528,820 routes through a 20×20 grid }
```

So there are `137,846,528,820` routes through a *20×20* grid. *{ 0.54" virtual, 2' 7" physical }*

Comments

Brute-force isn't an option here and though there are problems where the use of *recursion* results in a short, fast, clear and elegant solution (see ***PE#077*** and ***PE#093*** below) this *isn't* one of them. Attempting to use recursion:

```
10  DESTROY ALL @ OPTION BASE 1 @ N=0 @ L=1
20  FOR I=1 TO 20 @ CALL ROUTES(I,I,N,L) @ DISP I;N @ NEXT I

30  SUB ROUTES(A,B,N,L) @ IF A=1 THEN N=B+1 @ END ELSE IF B=1 THEN N=A+1 @ END
40  N=1 @ FOR I=1 TO A @ CALL ROUTES(I,B-1,M,L+1) @ N=N+M @ NEXT I
```

indeed results in a short, clear and elegant solution but certainly *not* fast: the run time grows *exponentially* with the grid size and by the time we reach a mere *15×15* (never mind *20×20)* we're talking *hours* or worse. Here the non-recursive solution clearly wins hands down (but see ***Note 1***).

Project Euler #017: ***Number letter counts***

*If the numbers **1** to **5** are written out in words: one, two, three, four, five, then there are 3+3+5+4+4=**19** letters used in total.*

*If all the numbers from **1** to **1000** (one thousand) inclusive were written out in words, how many letters would be used?*

NOTE: Do not count spaces or hyphens. For example, 342 (three hundred and forty-two) contains 23 letters and 115 (one hundred and fifteen) contains 20 letters. The use of "and" when writing out numbers is in compliance with British usage.

My solution for the ***HP-71B*** is[1]*:*

```
 10 DESTROY ALL @ OPTION BASE 1 @ DIM U$(19)[9],D$(9)[7] @ READ U$,D$ @ S=0
 20 DATA one,two,three,four,five,six,seven,eight,nine,ten,eleven,twelve
 30 DATA thirteen,fourteen,fifteen,sixteen,seventeen,eighteen,nineteen
 40 DATA ten,twenty,thirty,forty,fifty,sixty,seventy,eighty,ninety
 50 FOR I=1 TO 1000 @ S=S+LEN(REPLACE$(REPLACE$(FNN$(I)," ",""),"-","")) @ NEXT I
 60 DISP "Count:";S

100 DEF FNN$(N) @ IF N=1000 THEN FNN$=”one thousand” @ END ELSE N$=""
110 IF N<20 THEN FNN$=N$&U$(N) @ END
120 U=MOD(N,10) @ D=MOD(N DIV 10,10) @ C=N DIV 100 @ IF C THEN 140
130 N$=N$&D$(D) @ IF U THEN FNN$=N$&"-"&U$(U) @ END ELSE 150
140 N$=U$(C)&" hundred" @ IF U+D THEN N$=N$&" and " @ N=MOD(N,100) @ GOTO 110
150 FNN$=N$
```

RUN → `Count: 21124` *{ 0.84” virtual, 3’ 58” physical }*

So `21,124` letters would be used to write out the first *1,000* numbers in words.

Comments

My program first initializes a string array with the words for *0-19* (*“zero”*, *”one”*, ..., *“nineteen”*), another with the words for *0*, *10*, ..., *90* (*“zero”*, *”ten”*, ..., *“ninety”*) and then tallies the total number of letters by simply looping through all numbers from *1* to *1,000*, adding for each the number of letters of the equivalent wording returned by a *user-defined string function* **FNN$**, which essentially does all the work.

FNN$ accepts a numeric value as its argument and returns the equivalent wording by disassembling it into its units, decades and hundreds and reassembling the words for each component. The particular value *1,000* is singled-out early. It can be used in other programs or even called from the command line, like this:

```
>FNN$(25)      END LINE  →   twenty-five
>FNN$(8*13)    END LINE  →   one hundred and four
>FNN$(969)     END LINE  →   nine hundred and sixty-nine
>FNN$(517)     END LINE  →   five hundred and seventeen
>FNN$(550)     END LINE  →   five hundred and fifty
>FNN$(111)     END LINE  →   one hundred and eleven
```

[1] The code uses the ***REPLEX*** keyword **REPLACE$** to quickly delete spaces/hyphens from the wording returned by **FNN$**. If unavailable, either edit **FNN$** to not include spaces/hyphens in the wording (see ***Note 2***) or use this equivalent *BASIC* code:
- *add* line: ***200* DEF FNL(S$) @ L=LEN(S$) @ FOR J=1 TO L @ L=L-(POS(" -", S$[J,J])#0) @ NEXT J @ FNL=L**
- *change* lines *50* and *60* to just: ***50* S=0 @ FOR I=1 TO 1000 @ S=S+FNL(FNN$(I)) @ NEXT I @ DISP "Count:";S**
Alas, the program can be *halved* and run *faster* as well by *not using strings at all !*. Simply replace each string by its length everywhere and every string operation/variable by the equivalent numeric operation/variable. **FNN$** becomes **FNN**, etc.

Project Euler #040: ***Champernowne's constant***

An irrational decimal fraction is created by concatenating the positive integers:

0.123456789101112131415161718192021...

*It can be seen that the 12th digit of the fractional part is **1**.*

If d_n represents the nth digit of the fractional part, find the value of the following expression:

$$d_1 \times d_{10} \times d_{100} \times d_{1000} \times d_{10000} \times d_{100000} \times d_{1000000}$$

My *113-byte* solution for the ***HP-71B*** is[1]:

```
10  DEF FNP(N)=(IP(LGT(N))+1)*N-(10^(IP(LGT(N))+1)-1)/9+1
20  DEF FND(P) @ N=IP(FNROOT(1,1000000,FNP(FVAR)-P)+.001)
30  P=P-FNP(N)+1 @ FND=VAL(STR$(N)[P,P])
```

No need to run a program, simply evaluate this from the command line:

```
>DESTROY ALL  END LINE
>FND(1)*FND(10)*FND(100)*FND(1000)*FND(10000)*FND(100000)*FND(1000000)  END LINE

    210          {   0.15" virtual,    42" physical }
```

Comments

A little experimentation with a suitably long string version of the constant will easily produce these data:

Range	Starting position	Range of positions
1 – 9	p(n) = n	1 – 9
10 – 99	p(n) = 2*n-10	10 – 188 + 1
100 – 999	p(n) = 3*n-110	190 – 2,887 + 2
1,000 – 9,999	p(n) = 4*n-1,110	2,890 – 38,886 + 3
10,000 – 99,999	p(n) = 5*n-11,110	38,890 – 488,885 + 4
100,000 – 999,999	p(n) = 6*n-111,110	488,890 – 5,888,884 + 5

My code isn't a runnable program but instead uses the above data to help implement two numeric *user-defined* functions, **`FNP`** *(single-line)* and **`FND`** *(multi-line)*, which can be executed right from the command line.

1) **`FNP(N)`** returns the position ***P*** in the constant where the given value ***N*** begins. For instance:

```
>FNP(1);FNP(9);FNP(1000);FNP(9999);FNP(100000);FNP(999999)  END LINE

   1  9  2890  38886  488890  5888884
```

*{ **1** appears in the 1st position, **1000** in the 2,890th posit., etc. }*

2) **`FND(P)`** is the inverse of **`FNP(N)`**, given the position ***P*** in the constant it returns the digit ***D*** at that position. We use **`FND`** to find the individual digits at the given positions in *Champernowne's* constant:

```
>FND(1);FND(10);FND(100);FND(1E3);FND(1E4);FND(1E5);FND(1E6)  END LINE

   1   1   5   3   7   2   1
```

*{ so **1** appears at position 1, **5** at pos. 100, **7** at pos. 10,000, etc. }*

and their product is: *1×1×5×3×7×2×1=* `210` , as seen above.

[1] The code uses the ***Math ROM***'s keyword **`FNROOT`** to find a root of a non-polynomial equation so a *Math ROM* is required.

Project Euler #077: *Prime summations*

*It is possible to write **10** as the sum of primes in exactly **5** different ways:*

*7 + 3 = **10**, 5 + 5 = **10**, 5 + 3 + 2 = **10**, 3 + 3 + 2 + 2 = **10** and 2 + 2 + 2 + 2 + 2 = **10***

*What is the first value which can be written as the sum of primes in over **5000** different ways ?*

My *244-byte* solution for the ***HP-71B*** is:

```
10  DESTROY ALL @ OPTION BASE 1 @ DIM P(25) @ READ P
20  DATA 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97
30  FOR N=11 TO INF @ S=0 @ CALL PSUM(N,P,1,S) @ DISP N;S @ IF S>5000 THEN END
40  NEXT N

50  SUB PSUM(N,P(),K,S) @ FOR I=K TO 25 @ M=N-P(I) @ IF M<=0 THEN S=S+NOT M @ END
60  CALL PSUM(M,P,I,S) @ NEXT I
```

```
RUN  →  11   6      { 11 can be written in 6 ways as a sum of primes }
        12   7      { 12 can be written in 7 ways as a sum of primes }
          ...
        69   4268   { 69 can be written in 4,268 forms as a sum of primes }
        70   4624   { 70 can be written in 4,624 forms as a sum of primes }
        71   5007   { 71 can be written in 5,007 forms as a sum of primes }   {  8' 7" virtual,   38h physical }
```

So the solution is (quite fittingly) ***71***, which can be written as a sum of primes in *5,007* different ways.

Comments

Recursion has a bad reputation of being resource-consuming and slow (mainly because of examples like the dreadful *recursive* implementation of the ***Fibonacci*** series *vs* the iterative one), but in my experience it can really help *simplify* the implementation of many complex functionalities that can be coded in less lines using recursion and are easier to understand and debug as well. That's the case here (and also in ***PE#093*** below.)

My program begins by filling up an array ***P*** with the first 25 primes (enough for this problem) and then, starting with ***N****=11*, it calls a *recursive subprogram* **PSUM** which finds and returns in variable ***S*** the number of ways to write ***N*** as a sum of primes. Once the call returns, the main program simply displays both ***N*** and ***S***, and loops until ***S*** is over *5,000* , as required. In other words, the *2-line recursive* subprogram **PSUM** does *all* the work !

PSUM simply loops through the array of prime numbers starting at the *k-th* prime (where *k=1* for the first call), subtracting each prime from the value still remaining and doing a three-pronged check of the result:

(1) if the result is *< 0*, then this prime and the ones after it *exceed* the sum so it's not a valid way, return.

(2) if the result is *= 0*, then we have an *exact sum*, so increment the number of ways by one and return.

(3) if the result is *> 0*, then more primes are still needed, so it *recursively calls itself* with the new value to add up to and the index of the current prime just used (as it might be used multiple times). Upon returning from the recursive call, loop till all the primes have been considered and then return.

Note that the solving procedure is completely *general*, so you can use sequences other than the first 25 primes. For instance, editing the program to use instead this subset of the *Fibonacci* numbers *{1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377}* (just change the two *25* to *13* and the whole **DATA** statement at line *20*) we readily find that there are exactly *400 ways* to add up to *28* and more than *500 ways* (509 in fact) to add up to *30*.

Project Euler #093: ***Arithmetic expressions***

*By using each of the digits from the set, {1, 2, 3, 4}, exactly once, and making use of the four arithmetic operations (+, −, *, /) and brackets/parentheses, it is possible to form different positive integer targets.*

For example,

*8 = (4 * (1 + 3)) / 2, 14 = 4 * (3 + 1 / 2), 19 = 4 * (2 + 3) – 1, 36 = 3 * 4 * (2 + 1)*

Note that concatenations of the digits, like 12 + 34, are not allowed.

Using the set, {1, 2, 3, 4}, it is possible to obtain 31 different target numbers of which 36 is the maximum, and each of the numbers 1 to 28 can be obtained before encountering the first non-expressible number.

Find the set of four distinct digits, a<b<c<d, for which the longest set of consecutive positive integers, 1 to n, can be obtained, giving your answer as a string: abcd.

My *596-byte* solution for the ***HP-71B*** is*:*

```
 10  DESTROY ALL @ STD @ DEFAULT OFF @ DIM V$[512] @ R=0
 20  FOR A=1 TO 9 @ FOR B=A+1 TO 9 @ FOR C=B+1 TO 9 @ FOR D=C+1 TO 9 @ M=-1
 30  V$="#" @ CALL EXPR(STR$(A)&STR$(B)&STR$(C)&STR$(D),"",V$,M,0) @ K=M
 40  FOR I=1 TO M @ IF NOT POS(V$,"#"&STR$(I)&"#") THEN K=I-1 @ GOTO 60
 50  NEXT I
 60  IF K>R THEN R=K @ DISP A;B;C;D;":";K
 70  NEXT D @ NEXT C @ NEXT B @ NEXT A @ DEFAULT ON

 90  SUB EXPR(S$,N$,V$,M,P) @ M$="+-*/" @ FOR I=1 TO LEN(S$) @ T$=S$
100  E$=T$[I,I] @ D$=N$&E$ @ T$[I,I]="" @ IF T$="" THEN 140
110  FOR J=1 TO 4 @ Q$=M$[J,J] @ CALL EXPR(T$,D$&Q$,V$,M,(P))
120  IF P>0 THEN CALL EXPR(T$,D$&")"&Q$,V$,M,P-1)
130  CALL EXPR(T$,N$&"("&E$&Q$,V$,M,P+1) @ NEXT J @ NEXT I @ END
140  IF P>1 OR P<0 THEN END ELSE IF P=1 THEN D$=D$&")"
150  ON ERROR GOTO 170 @ N=VAL(D$) @ IF N<=0 OR FP(N) THEN END
160  M=MAX(M,N) @ IF NOT POS(V$,"#"&STR$(N)&"#") THEN V$=V$&STR$(N)&"#"
170  END SUB
```

```
[RUN]  →   1  2  3  4 : 28      {     31" virtual,    2h 28' physical }
           1  2  3  8 : 35      {  2' 36" virtual,   12h 22' physical }
             ...  ...
           1  2  5  6 : 43      {  6' 17" virtual,   29h 46' physical }
           1  2  5  8 : 51      {  7' 20" virtual,   34h 45' physical }
```

So the solution ***abcd*** is `1258` , which produces all consecutive integers from ***1*** to ***51***.

Comments

Once more, *recursion* proves extremely useful to efficiently implement a neat solution to a complex task like the one here, where the expressions to evaluate can be considered as formed by various *sub-expressions* to be evaluated *recursively*. Think for instance of something like the first example, *E=(4 * (1 + 3)) / 2*, which can be evaluated as ***E***=***U*** */ 2* where ***U***=*4 ** ***V*** where ***V***=*1+3*, all of them simpler *number-oper-number* sub-expressions.

Here the main program simply sets up four nested loops going in order through all possible values for ***a***, ***b***, ***c*** and ***d***, and within the innermost loop it calls the *recursive* subprogram **EXPR** which tries all combinations of the operations [+, −, *, /] and parentheses by *recursively calling itself* three times (to evaluate the sub-expressions), discarding those expressions which have *unbalanced* parentheses, produce *non-positive* or *non-integer* values or just *error out*, and recording (w/o repetitions) the valid results in a string, which the main program later checks to find out how many consecutive integers were produced and display on the go the currently best combinations.

Project Euler #094: *Almost equilateral triangles*

It is easily proved that no equilateral triangle exists with integral length sides and integral area. However, the almost equilateral triangle 5-5-6 has an area of 12 units.

We shall define an almost equilateral triangle to be a triangle for which two sides are equal and the third differs by no more than one unit.

Find the sum of the perimeters of all almost equilateral triangles with integral side lengths and area and whose perimeters do not exceed one billion (1,000,000,000).

My *186-byte* solution for the ***HP-71B*** is:

```
10  DESTROY ALL @ S=0 @ A=1 @ B=17 @ C=241 @ L=-1 @ GOSUB 30
20  A=1 @ B=5 @ C=65 @ L=1 @ GOSUB 30 @ DISP "Sum:";S @ END

30  S=S+3*(B+C)+2*L
40  D=15*(C-B)+A @ P=3*D+L @ IF P>1000000000 THEN RETURN
50  S=S+P @ DISP D,P,S @ A=B @ B=C @ C=D @ GOTO 40
```

```
RUN  →  3361        10082       10854         { area, perimeter and running sum of perimeters }
        46817       140450      151304
          ...          ...         ...
        2433601     7300804     416721290
        33895685    101687056   518408346

            Sum:  518408346                   {   ~0.01" virtual,   3.2" physical }
```

Comments

Trying to fully solve this problem by using a *brute-force* search is unbearably inefficient considering the *one billion* limit, but a very useful technique is to use simple brute-force up to a <u>much smaller limit</u> to get some data which can then be analyzed to detect *patterns*. For instance, this code for the case of the ***C*** side differing by ***–1***:

```
10 DESTROY ALL @ FOR A=1 TO 10000 @ S=FNA(A,A,A-1) @ IF NOT FP(S) THEN DISP A;A;A-1;S
20 NEXT A
30 DEF FNA(A,B,C)=SQRT((A+B+C)*(B+C-A)*(A+C-B)*(A+B-C))/4      { FNA  returns the area of  ΔABC }
```

when run produces these useful data:

```
RUN  →  1       1      0       0              { side A, side B=A, side C=A−1, integral area }
        17      17     16      120
        241     241    240     25080
        3361    3361   3360    4890480
```

and using my *LINREC* utility (see ***References***) to analyze the ***A*** sides we find this 3-term *linear recurrence*:

$$A_1 = \mathbf{1},\quad A_2 = \mathbf{17},\quad A_3 = \mathbf{241},\quad \text{and}\quad A_n = 15\,A_{n-1} - 15\,A_{n-2} + A_{n-3}$$

Editing and then running the above code for the ***+1*** case (just change the two ***–1*** in line *10* to ***+1***) produces instead the sequence ***1***, ***5***, ***65***, ***901*** for the ***A*** sides and using again *LINREC* we find this recurrence, also 3-term:

$$A_1 = \mathbf{1},\quad A_2 = \mathbf{5},\quad A_3 = \mathbf{65},\quad \text{and}\quad A_n = 15\,A_{n-1} - 15\,A_{n-2} + A_{n-3}$$

which is the *same* linear recurrence , only with different starting values. Using both recurrences in the program it reaches the *1 billion limit* extremely quickly and produces almost instantly the required sum of perimeters.

Project Euler #104: ***Pandigital Fibonacci ends***

The Fibonacci sequence is defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2} \quad , \quad \text{where } F_1 = 1 \text{ and } F_2 = 1.$$

It turns out that F_{541}, which contains 113 digits, is the first Fibonacci number for which the last 9 digits are 1-9 pandigital (contain all the digits 1-9, but not necessarily in order). And F_{2749}, which contains 575 digits, is the first Fibonacci number for which the first 9 digits are 1-9 pandigital.

*Given that F_k is the first Fibonacci number for which the first 9 digits AND the last 9 digits are 1-9 pandigital, find **k**.*

My *326-byte* solution for the ***HP-71B*** is[1]:

```
10 DESTROY ALL @ A=1 @ B=1 @ P=1 @ U=1 @ K=10^9 @ FOR I=3 TO INF
20 C=A+B @ X=P+U @ Y=Q+V+X DIV K @ Z=R+W+Y DIV K @ X=MOD(X,K)
30 Y=MOD(Y,K) @ IF Z<K THEN 50 ELSE X=Y @ Y=MOD(Z,K) @ Z=Z DIV K
40 U=V @ V=MOD(W,K) @ W=W DIV K @ P=Q @ Q=MOD(R,K) @ R=R DIV K
50 C=MOD(C,K) @ IF SPAN("123456789",STR$(C)) THEN 80
60 DISP I;C; @ H=10^IP(1+LOG10(Z)) @ H=Z*K DIV H+Y DIV H @ DISP H
70 IF NOT SPAN("123456789",STR$(H)) THEN DISP "K=";I @ END
80 A=B @ B=C @ P=U @ U=X @ Q=V @ V=Y @ R=W @ W=Z @ NEXT I
```

```
                 k        9 last digits of Fk   9 first digits of Fk

[RUN]  →    541        839725641     516212329     { 9 first digits of Fk aren't 1-9 pandigital }
            919        965324781     513046096     {    ditto   }
            ...           ...           ...
            328733     712489653     608775679     {    ditto   }
            329468     352786941     245681739     { Found!: 9 first and 9 last digits are 1-9 pandigital }

            K=  329468                             {   3' 23" virtual,   16h 2' physical }
```

So F_{329468} is the first ***Fibonacci*** number for which the *first* 9 digits *and* the *last* 9 digits are *1-9 pandigital*.

Comments

The very first thing is to realize that we *don't* need to compute *all* the digits of extremely large ***F*** numbers (the solution F_{329468} has *~69,000* digits), we just need the *first 9* and the *last 9*, which essentially makes the running time *linear* on ***k***, no matter how large F_k turns out to be. The latter are obtained by computing each term using the recurrence relation, adding the previous terms mod 10^9 and then checking the last 9 digits of the sum for *"1-9 pandigital-ness"* (the **SPAN** keyword is used to do it), displaying each successful candidate for feedback.

The former are also checked for *1-9 pandigital-ness* but if and only if the latter checked *Ok* (which saves a lot of time if they didn't), and could ideally be computed very quickly *if* we had *~17 digit* precision *(see **Note 3**)* but as we don't (*12-digit* only) we compute them accurately by keeping track of the first 27 digits for each F_k and periodically discarding the least significant 9 (so we're carrying 18 digits from F_k to F_{k+1}), using arithmetic modulo 10^9 for the additions. At check-time the first 9 are singled out (at line *60)* and checked out via **SPAN**.

If this second check also comes out Ok then we have a solution, which is output and the program ends. If any of the checks fail then the execution loops to generate and check the next candidate ***F*** number.

[1] The keyword **SPAN**, which is used to check if a number is 1-9 pandigital, is provided by the *STRINGLX LEX* file. If *STRINGLX* is not available but the *Math ROM* is, it can also be used to speed up the check somewhat, see ***Note 4***.

The Rant

Despite being awesome overall, ***Project Euler*** does also have its shortcomings which frustrated me frequently enough that I feel I must mention the main ones here, namely:

1) Far too many problems require using *more than 12 digits* to solve them, so they can't be solved using the *HP-71B* or any other 12-digit handheld computer or calculator even if using the best algorithms which otherwise could solve them quickly and efficiently. The only alternative would be to implement a sort of *"double-precision"* in *BASIC* but this usually results in a much longer, much slower program, it isn't practical and muddles everything so much that the problem utterly loses it's appeal (it can be done, though, if you're sufficiently motivated; see my sample *HP-71B* program for ***PE#104*** above).

For instance, ***PE #387 - Harshad Numbers*** can be solved with a short, clever recursive algorithm which was delightful to discover and implement but then it has a *15-digit* result, which forces my *71B* program to fall short of the mark at the intermediate 12-digit sum *497,168,223,439*, which is a pity.

Same thing happens with ***PE #196*** *(18 digits)*, ***PE #210*** *(19 digits)*, ***PE #214*** *(13 digits)*, ***PE #235*** *(13 digits)*, ***PE #276*** *(19 digits)*, ***PE #387*** *(13 digits)* and many others.

2) Again, far too many problems involve computing results for a given upper limit which frequently is *far too high* (and often results in more than 12 digits for the final result as well). Such excessively high limits also force extremely long execution times (we're talking many days) for the *71B*, so they can't be solved in reasonable time *even using the very best algorithms possible*. Again, incredibly frustrating.

This happens for some of the problems mentioned in *(1)* above and ***PE #162*** *(up to 16^{16})*, ***PE #168*** *(up to 10^{100})*, ***PE #171*** *(up to 10^{20})*, ***PE #193*** *(up to 2^{50})*, ***PE #601*** *(up to 4^{31})* and many others as well.

3) Some problems require the program *to read a sizable ASCII text file* as their input, e.g.: *5 Kb*, *10 Kb*, *16 Kb*, *26 Kb*, *46 Kb*, *...* . As trying to read these text files in an emulated *HP-71B* (let alone a *physical* one !!) is extremely cumbersome (never mind *manually* keying the data into the *71B)*, this makes it impractical or impossible to solve them using the *HP-71B* (though, again, it can be done if you're sufficiently motivated, as I very *tediously* did for ***PE #096***, successfully solving it with the *HP-71B*).

For example, this is the case for ***PE #022*** *(46 Kb)*, ***PE #054*** *(30 Kb)*, ***PE #081*** *(31 Kb)*, ***PE #082*** *(31 Kb)*, ***PE #083*** *(31 Kb)*, ***PE #098*** *(16 Kb)*, ***PE #102*** *(26 Kb)*, ***PE #424*** *(27 Kb)*, among others.

Of course I can understand that ***PE problems*** are aimed at being solved with *fast modern PC*/laptops (not ancient calculators, even if emulated) using *high-level compiled* languages (not interpreted *BASIC*), but that said I also feel that fulfilling ***PE*** goals, namely: *"... to provide a platform ... "*

"*...for the inquiring mind* ***to delve into unfamiliar areas*** *and* ***learn new concepts*** *in a* ***fun*** *and* ***recreational*** *context"*

can be achieved *without* resorting to artificially high limits and multiprecision results, as there's an infinite number of interesting problems fulfilling the stated goal without needlessly forcing such limiting constraints. There are much better, smarter ways to increase the difficulty, it's just a matter of finding them. Surely it will take more effort on the part of problem creators but ultimately it'll be much more rewarding for everyone than just taking the lame *"Hey, let's ask the result for n=$10^{zillion}$, that'll teach them !"* attitude.

The Results

By the end of August, 2011 the allotted fortnight had elapsed and I had solved ~***140 PE*** problems (the very first day I solved the first 25), achieved *Level 5* and was within the *15* topmost solvers in Spain (among ~*1,000* in all) and within the topmost *2,400* in the world (among ~*250,000* in all), i.e.: I was within the ***1%*** world percentile.

I also left another ~***50 PE*** problems *unfinished* at various states of completion, with plenty of notes and partial results (several text pages for each), due to sheer lack of time to investigate them any further, but the figures are irrelevant, the important thing to me was that I had extensively practiced and *enhanced* my math and programming skills and, as ***PE*** promised, my mind *did* delve into unfamiliar areas and I *learned* new concepts so ***PE***'s goal was achieved and mine too. And yes, it was *tremendously fun !* Highly recommended !!

Notes

1. ***PE #015***: as I originally attempted to solve each ***PE*** problem in ascending order from *PE #001* onwards, by the time I dealt with this *PE #015* I still hadn't developed all the techniques and tools that I did use for later problems, so for this one I didn't use the very useful technique of creating and running some simple procedure, even if grossly inefficient (brute force, recursion) up to a *much smaller limit* to quickly get data which I could then analyze to detect *patterns* (e.g.: some linear recursion) or even to consult *OEIS* for the particular short sequence obtained.

   Had I done so, running the simple *recursive* version would have produced the sequence *2*, *6*, *20*, *70*, *252*, *924* in less than a second, which *OEIS* readily identifies as sequence ***A000984*** *Central binomial coefficients: binomial(2*n,n)*, listing the value *137,846,528,820* for the 20th element, which is the solution. Furthermore, in the `COMMENTS` section it says: *"The number of direct routes from my home to Granny's when Granny lives n blocks south and n blocks east of my home in Grid City"*, which is fully equivalent to *PE*'s statement of the problem and kills it.

   Of course, using the information gathered at *OEIS* the *HP-71B*'s solution doesn't even require writing a program but reduces to evaluating this expression from the command line (requires the *JPC ROM* for the **`COMB`** keyword):

   ```
   >COMB(40,20)   ⟶   137846528820
   ```

   which is much simpler than any of my two attempts and instantaneously solves the problem, but I don't regret spending time concocting a solution without consulting references on the Internet, it was *way* funnier.

2. ***PE #017***: if *REPLEX* isn't available and you opt for *not* including spaces/hyphens in the wording returned by **`FNN$`** you must edit out the hyphen at line *140*, the spaces at lines *150* and *160*, and the space in *"one thousand"*. Also, change line *50* to: **`50 FOR I=1 TO 1000 @ S=S+LEN(FNN$(I)) @ NEXT I`**

3. ***PE #104***: if we had a high-precision *(~17-digit)* decimal logarithm at hand we could compute the first 9 digits by directly evaluating **`INT(10^(8+FP(K*LOG10((1+SQR(5))/2)-LOG10(5)/2)))`**, and for ***k=329468*** this gives ***245681739***, which indeed are the correct first 9 digits. However, when limited to the *HP-71B*'s native *12-digit* accuracy, the above expression evaluates to ***2456817<u>51</u>***, which isn't accurate enough. A real pity, as this forced me to keep a *27-digit* running total using modular arithmetic and culling, which is much, much slower.

4. ***PE #104***: If *STRINGLX* is not available but the *Math ROM* is, you can create this *UDF* to speed up the check:

   ```
   100 DEF FNP(N) @ S$=STR$(N) @ IF POS(S$,"0") THEN END ELSE IF LEN(S$)#9 THEN END
   110 MAT D=CON @ FOR J=1 TO 9 @ D(VAL(S$[J,J]))=0 @ NEXT J @ FNP=NOT RNORM(D)
   ```

   and then change in the main code these lines as indicated:

   ```
   10  DESTROY ALL ..                          to  DESTROY ALL @ OPTION BASE 1 @ DIM D(9)..
   50  .. IF SPAN("123456789",STR$(C)) ..      to  IF NOT FNP(C) ...
   70  .. IF NOT SPAN("123456789",STR$(H)) ..  to  IF FNP(H) ...
   ```

   If also unavailable, **`MAT..CON`** simply assigns ***1*** to all elements of ***D*** and **`RNORM`** adds up all the elements of ***D***.

References

https://projecteuler.net/about

Valentin Albillo	(2020)	*HP Article VA053 - Boldly Going - My Tools for Project Euler* *{ LINREC, etc. }*
Valentin Albillo	(2020)	*HP Article VA043 - HP-71B Advanced Uses of STRINGLX*
Valentin Albillo	(2005)	*HP Article VA011 - HP-71B Math ROM Bakers Dozen (Vol. 1)*
Valentin Albillo	(2005)	*HP Article VA012 - HP-71B Math ROM Bakers Dozen (Vol. 2)*

Copyrights

The following problems are taken from ***Project Euler**: **PE #015**, **PE #017**, **PE #040**, **PE #077**, **PE #093**, **PE #094**, **PE #104**.*