IOI'94 - Day 1 - Solution 3: The Primes

[Introduction] [Problem Statement] [Test Data]

Problem Analysis

Because *all* 5x5 squares of digits, satisfying certain requirements, must be generated (instead of just one), a systematic approach is necessary to exhaust all possibilities. One important subtask is to recognize five-digit primes with a given digit sum.

Let us investigate some numbers, such as the number of 5x5 digit squares. There are 25 digits in the square; the digit in the top left-hand corner is given. Thus, there are at most 10^24 squares to be considered. This number can be reduced considerably. For instance, the top row and the left column cannot contain any zeroes. Also the digit sums of five rows, five columns, and two diagonals are given, twelve digit sums altogether. That roughly means that twelve digits are determined by the others. Also observe that the nine digits in the bottom row and the rightmost column are odd. Forgetting about the details of primes, this means that the number of candidate 5x5 squares is on the order of 9^6*10^6/2^9, or approximately 10^9. This is more than you can expect to investigate in 90 seconds on a 33 MHz machine (90 seconds then give you 3*10^9 clock cycles).

Of course, many combinations can be ruled out at an early stage, because of the primality condition. We can either check primality of 5-digit numbers on-the-fly (while filling in the square), or make a table of them at the beginning (before filling in the square). What does it cost to check primality on-the-fly? Consider for instance the method that works by trial division. There are 65 primes <=sqrt(99,999). (The 65th and 66th prime are 313 and 317, furthermore 313*313=97,969 and 317*317=100,489. I cheated here: I asked Mathematica. You can also make a rough estimate by using the Prime Number Theorem.) This means that primality testing can be done by at most 65 divide operations. For primes it indeed takes at least that many divisions. This method seems too time-costly to do on-the-fly.

Generating Primes

We decide to pre-compute a table of 5-digit primes with a given digit sum. This is also useful because it will tell us how many such primes there are. In fact, we can make a file with all 5-digit primes together with their digit sum. Our square-generating program then starts off to read the primes with the appropriate digit sum from this file. Since this way the prime finding is kept outside the square generator, we do not have to worry too much about how we check primality. Here is a function IsPrime that works by trial division with all integers less than the square root (for n=99,999, this involves no more than 316 divisions):

```
function IsPrime(n: integer): boolean ; { pre: n >= 4 }
  var i, j, h: integer ;
  begin
  h := trunc(sqrt(n)) + 1 ;
  i := 2 ; j := h ; { bounded linear search for smallest divisor }
  while i <> j do
    if n mod i = 0 then j := i { i divides n, n is not prime }
    else inc(i) ;
  IsPrime := (i = h)
  end { IsPrime } ;
```

We will also count how many primes there are for each combination of digit sum and first digit. In order to get some confidence in this program, we would like to verify the results (do you know all 5-digit primes by heart; do you even know how many there are?). Therefore, we generalize the program to generate all primes with a given number ND of digits. The results for 2-digit primes can easily be verified by just looking at them. Also the 5-digit primes from the solution in the problem statement can be verified.

```
const
  ND = 5 ; { generate primes with ND digits }

var
  count: array[2..ND*9, 1..9] of integer ;
    { count[s, f] = # ND-digit primes with digit sum s and first digit f }
```

```
procedure Generate ;
  var first, i, s, f: integer ;
  begin
  first := 1 ;
  for i := 1 to pred(ND) do first := 10*first ; { first = 10^(ND-1) }
  for i := first to pred(10*first) do
    if IsPrime(i) then begin
      f := i ; s := i mod 10 ;
      while f >= 10 do begin f := f div 10 ; s := s + f mod 10 end ;
      { s = digit sum ; f = first digit }
      writeln(primes, i:ND, ' ', s:2) ;
      inc(count[s, f])
      end { if }
  end { Generate } ;
```

Program generate.pas generates the ND-digit primes in one file (e.g., primes-2.dat for ND=2 and primes-5.dat for ND=5) and the table of counts in another file (e.g., counts-2.dat for ND=2 and counts-5.dat for ND=5). From the table for ND=5 we learn that in total there are 8363 five-digit primes, and that digit sum 23 yields the most, namely 757 primes.

Generating Prime Squares

Let us introduce some notation. We number the rows of the square from top to bottom starting at 1, and the columns from left to right also starting at 1. The digit in row r and column c is denoted by S[r, c]. Here are some Pascal declarations:

```
type
  digit = 0..9 ;
  index = 1..5 ;

var
  ds: integer ; { given digit sum }
  tld: digit ; { given digit in top left-hand corner }
  S: array [index, index] of digit ; { the square }
```

We will fill and check the square systematically in twelve steps. Each step concerns a sequence of five digits, called a slot: there are five horizontal slots, five vertical slots, and two diagonal slots. During this process some positions in the square have been filled with a digit and others not yet. Whenever a new slot is considered, we fill it with all possible candidate primes from the table (one by one). Of course, such a prime has to match the digits that are already in the square. When all twelve slots have been filled with primes we have a solution. This programming technique is known as *backtracking*.

One thing that we have not discussed is the order in which we fill the slots. To reduce the number of possibilities as much as possible, we should try to fill slots for which as many digits as possible are already known, because this restricts the number of primes that will fit there. Many orders are possible. Let me introduce names for the slots as follows: H1 to H5 for the horizontal slots (rows 1 to 5), V1 to V5 for the vertical slots (columns 1 to 5), and D1 and D2 for the diagonal slots (D1 from top-left to bottom-right and D2 from bottom-left to top-right). We choose the order:

H1, V1, D2, H2, V2, H3, V3, H4, V4, H5, V5, D1

The digits are then produced in the following order:

```
 1  2  3  4  5
 6 13 14 12 15
 7 16 11 18 19
 8 10 20 22 23
 9 17 21 24 25
```

A nice property of this order is that the first digit is already known for every slot that is being filled: The top-left digit is given and it is the first digit of H1 and V1. The first digits of all (other) slots are covered by H1 and V1. Since the first digit of each slot is known, we can restrict attention to primes from the table that start with that digit. To exploit this we organize the table of primes as follows:

```
type
  number = record
    d: array [index] of digit ; { d[i] is i-th digit of v }
    v: integer { 5-digit prime with digit sum ds }
    end { number } ;

var
  n: integer ; { # primes }
  prime: array [0..800] of number ; { prime[0..n-1] filled in }
  first, last: array [digit] of integer ;
    { prime[i].d[1] = f for first[f] <= i <= last[f] }
```

The record `number' is introduced because we need to access the digits of each prime individually and also the `whole value' (the latter will become clear later on). The arrays `first' and 'last` indicate for each digit f the first and last index (in array `prime') of primes with first digit f. They are filled in when reading the primes from the file:

```
procedure ReadPrimes ;
  { read primes with digit sum ds from file }
  { pre: primes are sorted in increasing order }
  var primes: text ; p, s: integer ; i: index ; f: digit ;
  begin
  assign(primes, 'primes-5.dat') ; reset(primes) ;
  for n := 1 to 9 do begin first[n] := -1 ; last[n] := -2 end ; { empty ranges }  n := 0 ;
  while not eof(primes) do begin
    readln(primes, p, s) ; { read a prime p and its digit sum s }
    if s = ds then with prime[n] do begin
      v := p ;
      for i := 5 downto 1 do begin d[i] := p mod 10 ; p := p div 10 end ;
      if first[d[1]] = -1 then first[d[1]] := n ;
      last[d[1]] := n ;
      inc(n)
      end { if with }
    end { while } ;
  if Test then begin
    writeln('Number of 5-digit primes with digit sum ', ds:1, ' is ', n:1) ;
    writeln('Number of these primes with first digit f:') ;
    write('  f = ') ;
    for f := 1 to 9 do write(f:4) ;
    writeln ; write('  # = ') ;
    for f := 1 to 9 do write(last[f]+1-first[f]:4) ;
    writeln
    end { if }
  end { ReadPrimes } ;
```

At the end of the procedure we write out some test data to help us verify that reading from the file works all right (the test output can be compared to the table of counts from the prime generator).

All that now remains is to fill the slots. For each slot we write a procedure that has the obligation to fill that slot in all possible ways and for each successful filling to call the procedure dealing with the next slot. Procedure `Compute` that finds all solutions then simply becomes:

```
var
  solutions: integer ;

procedure ComputeAnswer ;
  begin
  S[1, 1] := tld ;
  solutions := 0 ;
  H1 ;
  if Test then writeln('Number of solutions is ', solutions:1)
  end { ComputeAnswer } ;
```

We will explain the procedures for some of the slots only, the others being very similar. For slot H1 we have

```
procedure H1 ;
  const R = 1 ;
  var i: integer ; c: index ;
```

```
  begin
  for i := first[tld] to last[tld] do with prime[i] do
    if d[2] <> 0 then
      if d[3] <> 0 then
        if d[4] <> 0 then begin
          for c := 2 to 5 do S[R, c] := d[c] ;
          V1
          end { if }
  end { H1 } ;
```

Only the first digit is known. All primes with this first digit are traversed by the for-loop. Only the primes without zeroes are filled in, after which V1 continues. The procedure for V1 is almost the same. For slot D2 we have

```
procedure D2 ;
  var i: integer ;
  begin
  for i := first[S[5, 1]] to last[S[5, 1]] do with prime[i] do
    if d[5] = S[1, 5] then begin
      S[4, 2] := d[2] ; S[3, 3] := d[3] ; S[2, 4] := d[4] ;
      H2
      end { if }
  end { D2 } ;
```

The first digit of slot D2 is in position S[5, 1]. For each prime with this first digit, we check whether its last digit d[5] matches the digit already present in position S[1, 5] (filled in by H1). If that is the case, the other three digits of the prime are copied into the square and H2 continues. Procedures H2, V2, H3, V3, and H4 are all very similar, only more digits are known and must be checked, whereas fewer digits are copied.

Procedure V4 is slightly different, in that all but the last digit are already known. We can simply compute the remaining digit because the digit sum is given. When this digit is computed, all that remains is to check whether the resulting number is prime. For that purpose we have introduced a function to look up a number in the table of primes. Since the table of primes is sorted on magnitude we can do a ***binary search*** (this is the reason why not only the digits of the primes are needed but also their `whole values'). Here is function IsPrime:

```
function IsPrime(w: integer): boolean ;
  { return: w is a 5-digit prime with digit sum ds }
  var i, j, h: integer ;
  begin
  i := 0 ; j := n ; { binary search }
  { w in prime[0..n-1].v == w in prime[i..j-1].v }
  while i <> pred(j) do begin
    h := (i+j) div 2 ;
    if prime[h].v <= w then i := h else j := h
    end { while } ;
  IsPrime := (prime[i].v = w)
  end { IsPrime } ;
```

The range of candidates can already be reduced because we know the first digit. However, the gain is not much (check the table of counts). You are challenged to try the following approach yourself and compare the execution times:

```
function IsPrime2(w: integer; f: digit): boolean ;
  { pre: f = first digit of w }
  var i, j, h: integer ;
  begin
  i := first[f] ; j := succ(last[f]) ; { binary search }
  if i >= j then IsPrime2 := false
  else { i < j } begin { w in prime[0..n-1].v == w in prime[i..j-1].v }
    while i <> pred(j) do begin
      h := (i+j) div 2 ;
      if prime[h].v <= w then i := h else j := h
      end { while } ;
    IsPrime2 := (prime[i].v = w)
    end { else }
  end { IsPrime2 } ;
```

In calls to IsPrime2 you have to supply the (known) first digit of the actual parameter for w. Note that we have to be careful: the range of primes with the required first digit may well be empty. [Story: My first program did not have the part `if i >= j then IsPrime2 := false' to guard for an empty range (because I naively modified IsPrime above). That program worked fine for all test data of the jury. I only found the mistake when trying the program for all possible input combinations (also see at the end). The first failure did not occur until digit sum 40. In all preceding cases there apparently never was the need to check primality with an empty range.] My advice: keep it simple.

Procedure V4 is now coded as follows:

```
procedure V4 ;
  const C = 4 ;
  var d, w: integer ; r: index ;
  begin
  d := ds ; w := 0 ;
  for r := 1 to 4 do begin
    d := d - S[r, C] ;
    w := 10*w + S[r, C]
    end { for } ;
  if odd(d) and (0 <= d) and (d <= 9) then
    if IsPrime(10*w+d) then begin
      S[5, C] := d ;
      H5
      end { if }
  end { V4 } ;
```

Procedure H5 is similar to V4. For slots V5 and D1 all digits are known and the only remaining task is to check primality. Here is procedure D1:

```
procedure D1 ;
  var w: integer ; rc: index ;
  begin
  w := 0 ;
  for rc := 1 to 5 do w := 10*w + S[rc, rc] ;
  if IsPrime(w) then WriteSolution
  end { D1 } ;
```

Altogether this yields Program 1. Once you have made some design decisions, this program is not difficult to write. There are, however, three thing that I dislike about it.

Firstly, it is easy to make a typing mistake that is very hard to find. The obligations of the twelve slot-filling procedures are crystal clear and it is just a matter of writing them. No problem. But it is easy to make a small mistake that is hard to spot afterwards: I made three typing errors. Don't kid yourself: you are not going to read those twelve procedures very carefully when under pressure. Lesson: programs that are easy to write are not necessarily easy to read. In a competition readability is important as well.

Secondly, when you do make a mistake (and this is quite likely), it may be easy to find out that there is a mistake *somewhere*. But for this program it is difficult to produce appropriate intermediate output to help diagnose errors. For instance, at each moment only a subset of the positions in the square contain valid digits (from slots already filled in). The other digits are junk, but you cannot see that by inspecting them. The knowledge of which digits are valid is encoded in the ***structure*** of the program, it is not encoded in its variables. Thus, it is difficult to print only the valid digits of a partially filled square.

Program 2 contains some diagnosing facilities. The main idea is to introduce an imaginary digit (undef=10) to mark an unfilled position (invalid digit). Each slot-filling procedure puts back `undef' after each attempt (it takes away the digits filled in). Procedure WriteSquare writes the valid digits of a square. This makes it possible to trace the filling process. (And even then it is still difficult to pinpoint errors.)

My third objection to Program 1 is that it is difficult to change the order in which slots are filled. The reason is that the choice of order is encoded, again, in the ***structure*** of the program. Of course, the slot-filling procedures can (relatively easily) be called in a different order (for instance, procedure `Compute` starts with D1, and D1 calls H1, etc.). But their obligations (and thus their program texts) also depend on that order. The filling order is quite critical in this problem. I believe that I made a reasonable choice, but you cannot always tell from the start. If you do find out that your program is too slow because you chose an inconvenient order, then you may want to

change that order, without major rewrites that can introduce mistakes.

For this problem, it is rather difficult to write a program that is both general (in that it allows you to change the filling order without major program modifications) and efficient. The point is that efficiency is precisely obtained by treating each slot in an optimal way, using knowledge about what is already filled in. If you have enough time, you can write a program that given an order generates an efficient Pascal program for generating squares based on that order.

Experiments

It is instructive to experiment with slight modifications in the program. For instance, how important is it to avoid zeroes in the top row and left column (in procedures H1 and V1). That is, how much slower (or faster?) becomes the program if you do not eliminate zeroes as early as possible? [N.B. Entries first[0] and last[0] are not defined in Program 1 because they are not needed. When zeroes are not directly eliminated, these entries become necessary.]

Also experiment with different orders. For instance, the order starting

H1, V1, H2, V2, ...

is interesting because the missing center digit of D2 can then be computed. When of the remaining 8 digits, two are chosen, the six others can be computed. All that then remains is to check primality of eight numbers.

Another experiment to carry out is generating the table of primes inside the program instead of reading it from a file.

All solution counts

We ran the program for all possible combinations of digit sum and top left-hand digit. The results are tabulated in file solcount.dat. Altogether there are 2676 prime squares. Top left-hand digit 3 and digit sum 23 yield the maximum number of 152 solutions. By the way notice the regular pattern of empty bands in the table of solution counts (there are only solutions for digit sums 11 and 13, 17 and 19. 23 and 25, 29 and 31, 35 and 37). Can you explain this pattern? (Let me know if you do, because I have not given it much thought. In September 1996, I received Mark Dettinger's explanation. Thank you, Mark!)

Variants of this problem

Of course, you can try this problem for squares of other dimensions than 5x5. How about more than two dimensions, for example, a 5x5x5 cube? What about other number systems than base 10?

Some combinations of top left-hand digit and digit sum have many solutions. Find the ones with the fewest number of different primes in the square. What is the smallest number of different primes to make a 5x5 prime square?

Find all rotation-invariant 5-digit primes, that is, find all 5-digit primes, all of whose rotations are prime as well. The four rotations of 12345 are 23451, 34512, 45123, and 51234.

Tom Verhoeff
Eindhoven University of Technology