Computer Science 301 - 2013

Programming Language Translation

Practical for Week 19, beginning 27 August 2013 - Solutions

The submissions received were very varied, but on the whole of rather reasonable quality. There was some innovative work handed in, but there was evidence that people had missed several important points. You can find complete source versions of the program solutions in the solution kit `PRAC19A.ZIP` on the server. This file also contains C# versions of the solutions for people who might be interested.

Some general comments:

(a) You should *always* put your names and a brief description of the program into the source code.

(b) Several submissions had almost no commentary at all, and this is just unacceptable. In particular, supply commentary at the start of each method as to what it sets out to do, and on the significance of the parameters/arguments.

(c) The pracs in this course are deliberately set to extend you, in the hope that you will learn a lot from each one. Their completion requires that you apply yourself steadily throughout the week, and not just on Thursday afternoon and the following Thursday morning!

(d) Some submissions were received that had not made proper use of the `IO`, `InFile` and `OutFile` classes *as you had been told to do*. These library classes are designed to make text based I/O as simple as possible without having to deal with buffered readers, exceptions, string tokenizers and all that stuff that you were probably subjected to in CSC 102, but without realizing that the best thing to do with bizarre code is to hide its details in a well-designed library. Have a look at the solutions below, where hopefully you will see that the I/O aspects have been made very simple indeed.

(e) Please learn to use the `LPRINT` facility for producing source listings economically. In later practicals the listings get very wide, and they are hard to read if they wrap round!

Tasks 2 to 7 - The Sieve of Eratosthenes

The first tasks were fairly straightforward, though several groups obviously had not bothered to see whether the extended sieve programs would execute properly. The Pascal compiler only uses 16-bit `INTEGER` arithmetic from (-32768 .. 32767), but it may appear to allow large array sizes, as arrays can also be indexed by so-called `long` variables. And, in fact (probably comes as a surprise to you C-language types), Pascal and Modula also allowed arrays to have negative indices, so that one could declare, for example

```
VAR RomePopulation       : ARRAY [-45 .. 320] OF INTEGER; (* an array with 366 elements *);
    BigArrayOfRealValues : ARRAY [0 .. 65534] OF REAL;    (* an array with 65535 elements *)
```

Since Pascal represented INTEGER values as signed 16-bit numbers, an extra limitation is imposed - an array indexed by an INTEGER variable cannot grant access to an element whose subscript is greater than 32767.

The question was, perhaps, badly worded. The Sieve size could get pretty large - in principle 65534 - but the Sieve algorithm could and did easily collapse when applied to large primes, using the standard INTEGER type. Consider the code:

```
K := I (* now cross out multiples of I *);
REPEAT
  Uncrossed[K] := FALSE; K := K + I
UNTIL (K > N);
```

When `I` becomes large enough, `K+I` should really be larger than 32767, but the overflow means that it appears to go negative (think back to your CSC 201 course). This happens for the first time after detecting the prime number 16411, so that the maximum effective sieve algorithm with the code above is limited to primes from 1 to 16411.

We can extend the range of the algorithm by a trick which I did not expect you to discover, but which may be worth pointing out. Replace the above code by

```
K := I (* now cross out multiples of I *);
REPEAT
  Uncrossed[K] := FALSE; K := K + I
UNTIL (K > N) OR (K < I)
```

which looks ridiculous, but when `K` gets too large and then overflows, since it appears to become negative it will then also appear to be less than `I`.

Much the same sort of behaviour happens in the 16-bit Modula-2 compiler. (Incidentally, the JPI Modula-2 compiler is much older (1989) than the Free Pascal one (currently 2013).) The type `CARDINAL` used in the Modula-2 code is implemented as an unsigned 16-bit number, so it might at first appear as before that we can use a sieve of about 65000 elements. However, when we reach the prime number 32771 the value of `K + I` overflows. This time we can still use a coding trick like that above, but we have to run the compiler in "optimized" mode to suppress the overflow detection that it normally provides. All rather subtle - up till now you probably have not really written or run code that falls foul of overflow or rounding errors, but they can be very awkward in serious applications.

Some students get intrigued by all this and probe further (well done). If you try interesting things like "turn off the range checks" the algorithms appear to allow you to generate higher prime numbers. Trouble is, they don't do it properly, and you find that for some "bigger" sieve arrays you actually seem to get fewer prime numbers generated.

Frankly, learning to program in "non-bondage" languages like C++ is like trying to learn to drive in a car without brakes - very exciting, you go faster and faster, and then you die, sooner or later. Fortunately C# and Java are much safer.

The 32-bit compilers don't seem to have this problem (or at least, it would be much harder to reproduce it), but, of course, the amount of real memory available to them may be limited).

There were several specious reasons thought up to explain why the executables were of such differing sizes. It is not true that this is a function of the sieve size to any marked degree, as the code and data areas are handled separately. The real reasons are that the efficiency of code generation differs markedly from one compiler to another, and that some implementations build in a great deal more extraneous code than others - you can see this in the smaller executable when some compilers are run in "optimizing" mode. The C and C++ executables differ enormously in size - no doubt due to the vast amounts of code needed to support the `iostream` library - and the Turbo Pascal 6.0 compiler produces amazingly tight code.

The Borland 5.5 and WatCom C/C++ compilers are 32-bit ones, rather than 16-bit ones. But even allowing for this, they suffer from bizarre code bloat for small applications. There are command line parameters and options that one can set to try to produce tighter code, if one bothers to experiment further. In fact, the scripts in the prac kit were not quite correct, as I discovered later - my apologies. Some (most) of the overheads may relate to the fact that they have to produce "Windows" compatible programs. Compilers like this are clearly designed with an "everyone has 8GB of memory and 3TB of disk space, and if they don't they should go and buy more" philosophy.

The "executables" produced from the C# compiler are not true executables in the same sense as those produced by the C++ and Pascal compilers, as the code in them still has to be "jitted" into its final form. Interestingly, an earlier release a few years ago produced sizes of 28 672, 28 672 and 16 896 for these programs.

The limitation imposed by the Parva system on the sieve size is entirely due to the fact that the interpreter system only allows for about 50 000 words of simulated memory - which has to hold the pseudo-code and all variables. The limit on the sieve size was a bit over 49 000. This could have been extended simply by modifying the interpreter, and then recompiling it, but you were not in a position to do that.

Execution times were slightly different, however.

	Sieve Code Size	Queens Code Size	Empty Code Size			
Turbo Pascal 1	11 751	12 246	11 386	Sieve limit 16410	.COM file sizes	DosBox
Turbo Pascal 6	2 912	3 728	1 472	Sieve limit 16410	.EXE file sizes	DosBox
Optimized Turbo Pascal 6	2 848	3 440	2 640	Sieve limit 16410	.EXE file sizes	DosBox
Free Pascal	34 612	31 812	30 516	Sieve limit 16410	.EXE file sizes	
Optimized Free Pascal	34 100	31 300	30 516	Sieve limit 16410	.EXE file sizes	
Modula-2	18 609	18 774	11 946	Sieve limit 32770	.EXE file sizes	DosBox
Optimized M-2	18 549	18 774	11 946	Sieve limit 32770	.EXE file sizes	DosBox
Borland C	66 560		52 224		.EXE file sizes	
Borland C++	149 504		47 104		.EXE file sizes	
Watcom C	34 816		21 504		.EXE file sizes	
Watcom C++	50 688		21 504		.EXE file sizes	
C#	32 256	45 056	31 744		.EXE file sizes	
Parva	N/A	N/A	N/A	Sieve limit 49000		
Modula-2 via Borland C	62 976		62 464		.EXE file sizes	

The Sieve in Parva

```
void main() {
// Sieve of Eratosthenes for finding primes 2 <= n <= 49000 (Parva version)
// P.D. Terry,  Rhodes University, 2013
  const Max = 49000;
  bool[] Uncrossed = new bool[Max];         // the sieve
  int i, n, k, it, iterations, primes = 0;  // counters
  read("How many iterations? ", iterations);
  read("Supply largest number to be tested ", n);
  if (n > Max) {
    write("n too large, sorry");
    return;
  }
  it = 1;
  while (it <= iterations) {
    primes = 0;
    write("Prime numbers between 2 and " , n, "\n");
    write("-----------------------------------\n");
    i = 2;
    while (i <= n) {                          // clear sieve
      Uncrossed[i-2] = true;
      i = i + 1;
    }
    i = 2;
    while (i <= n) {                          // the passes over the sieve
      if (Uncrossed[i-2]) {
        if (primes - (primes/8)*8 == 0)
          write("\n");                        // ensure line not too long
        primes = primes + 1;
        write(i, "\t");
        k = i;                                // now cross out multiples of i
        Uncrossed[k-2] = false;
        k = k + i;
        while (k <= n) {
          Uncrossed[k-2] = false;
          k = k + i;
        }
      }
      i = i + 1;
    }
    it = it + 1;
    write("\n");
  }
  write(primes, " primes");
} // main
```

The corrected code is very simple. A few groups got it badly wrong, with braces in the wrong places. Note that the body of a *do-while* loop has to be executed at least once, which means that the code should really have been transformed to achieve this. However, even if this is not done it "works". Why?

Task 8 - The N Queens problem

It was very easy to discover how many solutions can be found to the N Queens problem. Some people noticed that the number of distinctly different solutions would have been lower, as the total included reflections and rotations as though they were distinct.

2	3	4	5	6	7	8	9	10	11	12
0	0	2	10	4	40	92	352	724	2680	14200

Clearly the number of solutions rises very rapidly. Some groups rashly put forward unjustified claims that the number of solutions rose "exponentially". If you were very keen you might like to try to estimate the *O(whatever)* behaviour rather better from these figures (how?).

Task 9 - High level translators

Several students complained that the C code generated by the X2C system was "unreadable", and "not what we would have written ourselves". There can be no dispute with the second of those arguments, but if you take a careful look at the generated C code, it is verbose, rather than unreadable, because long identifier names have been used. This is actually not such a bad thing - at least one can tell the "origin" of any identifier, since the original module in which it was declared is incorporated into its name, and this is is a very useful trick, both for large programs, and (more especially) when one has to contend with the miserable rules that C has for controlling identifier name spaces. In fact, in the days when I used Modula regularly, I used a convention similar to this of my own accord, and have carried it over into my other coding, as you will see in several places in the book. Some of the other "unreadability" presumably relates to the fact that the X2C system is obliged to translate the `CARDINAL` type to the `unsigned int` type, which is one some of you will never have used - this explains all those funny casts and capital Us that you saw. Some people commented that "maintaining the C code generated would be a nightmare". Well, maybe, but the point of using a tool like this is that you can develop and maintain your programs in Modula-2 and then simply convert them to C when you want to get them compiled on some other machine. So normally a user of X2C would not read or edit the C code at all.

Task 9 - How fast/slow are various implementations?

Some times (seconds) taken to execute the various programs are shown below. for running these systems over several years, computers and operating systems. In earlier times the systems ran Windows XP-32. I suspect that not all these times are accurate, but we don't have the original systems to run more trials.

We note several points of interest:

(a) The Parva interpreted system is about two orders of magnitude slower than the native-code systems. Even here we can see the benefits of using an optimizing system, simple though this is (later we shall discuss this in more detail).

(b) In deriving these figures some experimentation was first needed so as to find parameters that would yield times sufficiently long to make comparisons meaningful. The times were obtained by simple use of a stopwatch, and of course there is always an element of reaction time in such measurements which we should minimize. The output statements were commented out so that all that was really being measured was the time for the algorithms themselves (I suspect not everybody realized this, as some submitted timings were very way out).

(c) The Java system, when JITted, is way better than when the JVM runs in pure interpreter mode.

(d) Even allowing for the reaction time phenomenon, there are some anomalies here. One might expect the execution times to be very closely related to the processor clock speeds, and that the ratio of times

measured on the laptops and the lab machines for each application would have been the same, but clearly they are not. I expect that the differences - which are quite marked - can be put down to the different interior architectures of the processors themselves, but I have not had time to explore this further.

Sieve: Iterations 10 000 Size of Sieve 16 000						
	1GHz LT	2GHz LT	3GHz PC	3GHz i7	3GHz I5	3GHz I5
	XP 32	XP 32	XP 32	XP 32	Win7 64	DosBox
Turbo Pascal 1.0		2.90		3.2		73
Turbo Pascal 6.0		43.70		25.4		500
Turbo Pascal 6.0 (opt)		3.23		3.0		69
Free Pascal	9.2	6.09	4.4	4.1		
Free Pascal (optimized)	8.6	4.57	4.2	3.4		
Modula-2	5.7	2.60	2.4	1.4		82
Modula-2 (optimized)	2.3	1.17	1.8	0.8		35
C	3.8	1.54	2.2	1.2	1.1	
C++	4.1	1.55	2.1	1.1	1.0	
Modula-2 (via C)	25.2	6.04	6.3	1.4	3.5	
C#	3.7	1.72	3.2	1.6	1.2	
Java (with JIT)	6.6	1.55	3.0	1.2	1.0	
Java -Xint (interpret)	74.3	10.64	46.6	11.4	9.0	
Parva	1080	475	448	335	240	
Parva (optimized)	810	375	360	275	210	

(e) The times taken by the 16-bit systems running under the DOSBox emulator show quite clearly the adverse effects of emulation. This system was apparently developed to allow game freaks to run "old" computer games designed in the 8086, 80386 and 80486 era to run on modern computers running at vastly greater clock speeds. There must be better ways of running old number-crunchers on the latest operating systems, and this will be investigated further next year.

Queens: Iterations 100	1GHz LT	2GHz LT	3GHz PC	3GHz i7	3GHz I5	3GHz I5
Board Size 11	XP 32	XP 32	XP 32	XP 32	Win7 64	DosBox
Turbo Pascal 1.0	5.5	11.15	3.2	6.10		
Turbo Pascal 6.0	5.5	27.44	3.2	16.22		175
T/Pascal 6.0 (optimized)	5.5	6.04	3.2	3.76		48
Free Pascal	5.5	3.75	3.2	2.34	2.20	
Free Pascal (optimized)	5.7	2.67	3.1	2.29	1.70	
Modula-2	9.2	5.60	7.5	3.54		144
Modula-2 (optimized)	8.0	5.30	5.8	3.52		48
C						
C++						
Modula-2 (via BCC)	11.0		4.6			
C#	3.8	2.33	2.9	1.67	1.52	
Java (with JIT)	5.4	2.23	2.6	1.45	1.45	
Java -Xint (interpret)	33.3	9.20	23.6	6.52	5.67	
Parva	434	185.0	178.0	123.00	127.0	
Parva (optimized)	324	145.0	139.0	102.0	97.0	

Queens1: Iterations 100	1GHz LT	2GHz LT	3GHz PC	3GHz i7	3GHz I5	3GHz I5	
Board Size 11	XP 32	XP 32	XP 32	XP 32	Win7 64	DosBox	
Turbo Pascal 1.0	5.2	3.00	3.1	1.90			
Turbo Pascal 6.0	5.2	24.50	3.1	16.30		162	
T/Pascal 6.0 (optimized)	5.2	3.34	3.1	2.31		38	
Free Pascal	5.2	3.80	3.1	2.34	2.20		
Free Pascal (optimized)	5.0	2.70	2.9	2.29	1.63		
Modula-2	7.4	4.80	6.6	2.21			redo
Modula-2 (optimized)	5.6	4.34	4.3	2.11			
C							
C++							
Modula-2 (via BCC)	10.7		4.3				
C#	4.0	2.23	2.8	1.57	1.59		
Java (with JIT)	5.3	2.47	2.8	1.62	1.70		
Java -Xint (interpret)	37.8	12.58	27.2	9.59	7.53		
Parva	425	183	186.0	126.0	120.0		
Parva (optimized)	326	147	139.0	99.0	93.0		

Of course, it is fairly dangerous to draw conclusive results from such a crude set of tests and limited sample, but the main effects show up quite clearly.

Task 11 - Something more creative - Play Sudoku

The solutions submitted were of very mixed quality. There were a few that I felt demonstrated excellent and enviable maturity in approach and coding skills. But there were a few that were still very incomplete, some that wrere completely confused, and some that showed that their authors prefer cutting and pasting code dozens of times in the editor rather than thinking through the simple mathematics required for a clean solution.

Here is code for the system I wrote, which derives the possibilities from the evidence, and then applies these systematically. Take a careful look at how I have tried to use the sets to best advantage and to guard against user errors - numbers out of range, numbers no longer assignable and so on.

This system will not solve all the puzzles in the kit. I am sure it could be improved still further. If you want to experiment further, look in the solution kit.

```
// Simple Sudoku helper - some hints given, and use made of the fact that each number
// must appear in each of the 3 x 3 sub matrices.
// all hints are optionally applied (when computer effectively plays as far as it can).
// The rules of Sudoku can be found in many on-line articles.
// PD Terry, Rhodes University, August 2013  ( 2013/09/16 )

import java.util.*;
import library.*;

class Sud5 {

  static final int SIZE = 9;

  static int count = 0, hints;
         // count of the numbers of moves that one can make based on the analysis

  static IntSet range = new IntSet(0, 1, 2, 3, 4, 5, 6, 7, 8);
         // acceptable row and column numbers for input (using 0 for quit)

  static IntSet all = new IntSet(1, 2, 3, 4, 5, 6, 7, 8, 9);
         // a fully populated set of permissible numbers in a cell prior to selection

  static IntSet [] [] possible = new IntSet[9][9];
         // the grid of sets of possible numbers still to be selected from,
```

```
        // one set for each of the 9 x 9 elements of the board

  static IntSet [] [] subMatrix = new IntSet[3][3];
        // the 3 x 3 grid of the submatrices, each of which must have each
        // of the numbers 1 ... 9 in it exactly once

  static int [] [] known = new int[9][9];
        // the solution board - populated as numbers are selected from the
        // possible sets for each of its 9 x 9 elements

  static int [] [] hint = new int[9][9];
        // the numbers predicted for each cell as the analysis proceeds are
        // stored here prior to a sweep across the board which stores those
        // numbers in the solution matrix and eliminates them from the old
        // possible sets whence thaye were selected

  static boolean choicesRemain;
        // true if further progress seems possible (it isn't always)

// ---------------------------------------------------------------------------- main

  public static void main(String[] args) {
    int row, col, n;
                                          // attempt to open data file
    if (args.length < 1) {
      IO.writeLine("Usage: Sudoku dataFile [-a]");
      System.exit(1);
    }

    InFile data = new InFile(args[0]);
    if (data.openError()) {
      IO.writeLine("cannot open " + args[0]);
      System.exit(1);
    }

    boolean automatic = args.length == 2 && args[1].toUpperCase().equals("-A");

    for (row = 0; row < SIZE; row++)          // initialize the possible matrix
      for (col = 0; col < SIZE; col++)
        possible[row][col] = (IntSet) all.copy();   // note that we must have 81 copies of the "all"
                                                    // set, and not 81 links to only one copy (easy
                                                    // mistake to make)

    for (row = 0; row <= 2; row++)            // initialize 3 x 3 submatrices
      for (col = 0; col <= 2; col++)
        subMatrix[row][col] = new IntSet();

    for (row = 0; row < SIZE; row++)          // read in the initial known matrix
      for (col = 0; col < SIZE; col++) {
        n = data.readInt();
        if (n != 0)                           // check for self-consistency
          if (possible[row][col].contains(n))
            claim(row, col, n);               // and nail that cell down to a known value
          else {                              // we blew it, inconsistent data
            IO.writeLine("Impossible data (" +  n + ") - line " + row + " column " + col);
            System.exit(1);
         }
      }

    showCurrentStateAndPredict();             // display initial matrices and a board of
                                              // known values and further predictions

    // now attempt to solve the puzzle further

    do {                                      // one analysis
      if (hints != 0 && automatic) {          // possibly incorporate hints - machine does work
        IO.writeLine("Press any key to apply all predictions"); IO.readLine();
        for (row = 0; row < SIZE; row++)
          for (col = 0; col < SIZE; col++)
            if (hint[row][col] != 0) claim(row, col, hint[row][col]);
        showCurrentStateAndPredict();
      }
      else {                                  // human move rather than letting the machine go for it
        IO.writeLine("Your move - row [0..8] col [0..8] value [1..9] (0 to give up)? ");
        row = IO.readInt();                      // just use the IO library - no need to split/parse
        col = IO.readInt();                      // and fight exceptions!
        n = IO.readInt();
        if (n == 0) System.exit(0);
        if (range.contains(row)                  // valid row
            && range.contains(col)               // valid column
            && possible[row][col].contains(n)) { // check that it is available
```

```
          claim(row, col, n);
          showCurrentStateAndPredict();
        }
        else IO.writeLine("******* Impossible");
      }
    } while (count != 81 && choicesRemain);
    if (!choicesRemain) IO.writeLine("\nNo more moves possible");

  }  // main

// ------------------------------------------------------------------------- claim

  static void claim(int row, int col, int n) {
  // Enters n into the known matrix, eliminates it from rows and columns of the possible
  // matrix and incorporates it into the associated submatrix as well
    known[row][col] = n; count++;
    for (int r = 0; r < SIZE; r++) possible[r][col].excl(n);
    for (int c = 0; c < SIZE; c++) possible[row][c].excl(n);
    subMatrix[row/3][col/3].incl(n);
    possible[row][col] = new IntSet();  // no longer only possible - we have it nailed
  } // claim

// ------------------------------------------------------------------------- showCurrentStateAndPredict

  static void showCurrentStateAndPredict() {
  // Incorporates the effects of the 3 x 3 submatrices and
  // displays the matrix of possible values, and the matrix of already known values in which
  // are embedded the suggestions the analysis has made

    int row, col, subRow, subCol;

    // eliminate elements from the possible matrix that are already members of a submatrix

    choicesRemain = false;
    for (row = 0; row < SIZE; row++)
      for (col = 0; col < SIZE; col++) {
        hint[row][col] = 0;
        possible[row][col] = possible[row][col].difference(subMatrix[row/3][col/3]);
        if (!possible[row][col].isEmpty()) choicesRemain = true;
      }

    IO.writeLine("Still possible before the next round of analysis");
    IO.writeLine("        0   1   2     3   4   5     6   7   8   ");
    IO.writeLine("     |=============+=============+=============|");
    for (row = 0; row < SIZE; row++) {  // all a bit messy, just to get neatly formatted output
      for (subRow = 0; subRow <= 2; subRow++) {
        if (subRow == 1) {
          IO.write(row, 3);
          IO.write("  | ");
        } else
          IO.write("     | ");
        for (col = 0; col < SIZE; col++) {
          for (subCol = 0; subCol <= 2; subCol++) {
            int n = subRow*3 + 1 + subCol;
            if (possible[row][col].contains(n))
              IO.write(n, 1);
            else
              IO.write(" ");
          };
          if ((col + 1) % 3 == 0)
            IO.write(" | ");
          else
            IO.write(".");
        };
        IO.writeLine();
      };
      if ((row + 1) % 3 == 0)
        IO.writeLine("     |=============+=============+=============|");
      else
        IO.writeLine("     |----+---+----+----+---+----+----+---+----|");
    };
    IO.writeLine();

    predictor();

    IO.writeLine("Already known - and with predictions in parentheses\n");
    IO.writeLine("       0   1   2   3   4   5   6   7   8");
    IO.writeLine();

    for (row = 0; row < SIZE; row++) {
      IO.write(row, 3); IO.write(": ");
```

```
      for (col = 0; col < SIZE; col++) {
        if (known[row][col] == 0)
          if (hint[row][col] != 0)
            IO.write(" (" + hint[row][col] + ")");
          else
            IO.write(" .. ");
        else {
          IO.write(known[row][col], 3);
          IO.write(" ");
        }
      }
      IO.writeLine();
    }
    IO.writeLine();
    IO.writeLine("       0   1   2   3   4   5   6   7   8");
    IO.writeLine();
    IO.write(count); IO.writeLine(" squares known. " + hints + " predictions");
  } // showCurrentStateAndPredict

// ------------------------------------------------------------------------- predictor

  static void predictor() {
  // Predicts and stores the values of obvious and not so obvious cells that can be deduced,
  // stores these in the hints matrix, and counts these hints.

  // Note that we only record one hint at most for each of the cells on the board,

    hints = 0;

  // look for "singles"
    for (int row = 0; row < SIZE; row++)
      for (int col = 0; col < SIZE; col++)
        if (possible[row][col].members() == 1)                  // only one possibility remaining - easy to pick
          for (int i = 1; i <= 9; i++)                          // sweep across to find which of 1..9 it was!
            if (possible[row][col].contains(i)) {
              hint[row][col] = i;
              hints++;
            }

  // look for numbers that appear only once in a row
    for (int row = 0; row < SIZE; row++)
      for (int i = 1; i <= 9; i++) {                            // sweep across to find which of 1..9 it was!
        int inRow = 0, r1 = 0, c1 = 0;
        for (int col = 0; col < SIZE; col++)
          if (possible[row][col].contains(i)) {
            inRow++;
            r1 = row;                                           // remember where you spotted it
            c1 = col;
          }
        if (inRow == 1 & hint[r1][c1] == 0) {                   // so you can set up the hint easily
          hint[r1][c1] = i;
          hints++;
        }
      }

  // look for numbers that appear only once in a column
    for (int col = 0; col < SIZE; col++)
      for (int i = 1; i <= 9; i++) {                            // sweep across to find which of 1..9 it was!
        int inCol = 0, r1 = 0, c1 = 0;
        for (int row = 0; row < SIZE; row++)
          if (possible[row][col].contains(i)) {
            inCol++;
            r1 = row;                                           // remember where you spotted it
            c1 = col;
          }
        if (inCol == 1 & hint[r1][c1] == 0) {                   // so you can set up the hint easily
          hint[r1][c1] = i;
          hints++;
        }
      }

  // look for "hidden singles" that appear only once in a sub-matrix
    for (int row = 0; row <= 2; row++)
      for (int col = 0; col <= 2; col++) {
        IntSet unused = all.difference(subMatrix[row][col]);    // here is the clever bit...
        for (int i = 1; i <= 9; i++)                            // sweep across to find which of 1..9 it was!
          if (unused.contains(i)) {
            int inBox = 0, r1 = 0, c1 = 0, keep = 0;
            for (int r = 0; r <= 2; r++)
              for (int c = 0; c <= 2; c++)
                if (possible[3*row + r][3*col + c].contains(i)) {
```

```
                    inBox++;
                    keep = i;
                    r1 = 3*row + r;
                    c1 = 3*col + c;
                  }
              if (inBox == 1 && hint[r1][c1] == 0) {
                hint[r1][c1] = keep;
                hints++;
              }
            }
          }
        } // predictor

}  // Sud5
```