Computer Science 301 - 2010

Programming Language Translation

Practical for Week 19, beginning 23 August 2010 - 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 realising 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 listing 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 (-32768 .. 32767) but it appears to allow large sieve sizes, as arrays can also be indexed by so-called `long` variables. Since it regards an integer as a signed 16-bit number, an extra limitation is imposed - an array indexed by an integer cannot have an upper subscript greater than 32767. But it's more complicated than that. 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` really should become 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 with the code above is really only 16410.

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 (2005).) The type `CARDINAL` used in the Modula-2 code is implemented as an unsigned 16-bit number, so it might at first appear 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.

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 is 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 could see this in the smaller executable when some compilers were 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.

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 may be command line parameters and options that one can set to try to produce tighter code, but I have not bothered to experiment further. 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 2GB of memory and 500GB 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, they are all of the same size with the current C# compiler. 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 50000 words of simulated memory - which has to hold the pseudo-code and all variables. The limit on the sieve size was a bit over 49000. This could have been extended simply by modifying the interpreter, and then recompiling it, but you were not in a position to do that.

Interestingly, it does not seem to make a difference if the Free Pascal compiler you were using is used in optimizing mode or not. An earlier release used in previous years was quite different - for example the Sieve program compiled to 15872 bytes in ordinary mode and to 14848 bytes in optimized mode.

	Sieve Code Size	Fibo Code Size	Empty Code Size	
Free Pascal	31 744	31 232	28 672	Sieve limit 16410
Optimized Free Pascal	31 744	31 232	28 672	Sieve limit 16410
Modula-2	18 609	18 098	11 946	Sieve limit 32770
Optimized M-2	18 549	18 049	11 946	Sieve limit 32770
Borland C	66 560	66 048	52 224	
Borland C++	149 504	148 480	47 104	
Watcom C	37 376	36 864	27 648	
Watcom C++	51 712	50 688	22 528	
C#	45 056	45 056	45 056	
Parva	N/A	N/A	N/A	Sieve limit 49000
Modula-2 via Borland C	62 976	62 464	62 464	
Modula-2 via Watcom	43 008	42 996	41 984	

The Sieve in Parva

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 had to be transformed to achieve this.

```
void main() {
// Sieve of Eratosthenes for finding primes 2 <= n <= 49000 (Parva version)
// P.D. Terry,  Rhodes University, 2010
  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");
}
```

Task 8 - 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 (figures in brackets are for a 1.04GHz laptop, others on 3.00 GHz lab machines a year or two ago). In all cases the systems ran Windows XP.

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 realised this, as some submitted timings were very way out).

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

(d) Even allowing for the reaction time phenomenon, there are some strange 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 laptop 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			
Fibo: Upper limit 40				
	Sieve		Fibo	
Free Pascal	(9.2)	4.4	(12.7)	4.2
Free Pascal (optimized)	(8.6)	4.2	(12.3)	4.1
Modula-2	(5.7)	2.4		
Modula-2 (optimized)	(2.3)	1.8	(23.5)	32.3
C	(3.8)	2.2	(12.3)	3.9
C++	(4.1)	2.1	(12.2)	4.0
Modula-2 (via C)	(25.2)	6.3	(18.9)	3.9
C#	(3.7)	3.2	(9.8)	5.1
Java (With JIT)	(6.6)	3.0	(9.3)	2.7
Java -Xint (interpret)	(74.3)	46.6	(92.6)	48.2
Parva	(1080)	448.0	(692)	277.0
Parva (optimized)	(810.0)	360.0		225.0

(I must have this wrong?)

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
// All hints are optionally applied (when computer effectively plays as far as it can)
// PD Terry, Rhodes University, August 2010

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

class Sud5 {

  static final int SIZE = 9;
  static int count = 0, hints;
  static IntSet range = new IntSet(0, 1, 2, 3, 4, 5, 6, 7, 8);
  static IntSet all = new IntSet(1, 2, 3, 4, 5, 6, 7, 8, 9);
  static IntSet [] [] possible = new IntSet[9] [9];
  static IntSet [] [] subMatrix = new IntSet[3] [3];
  static int [] [] known = new int[9] [9];
  static int [] [] hint = new int[9] [9];
  static boolean choicesRemain;

  static void predictor() {
  // Predicts hints to obvious and not so obvious cells that can be fixed and counts these

    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)
          for (int i = 1; i <= 9; i++)
            if (possible[row][col].contains(i)) { hint[row][col] = i; hints++; }

  // look for numbers that only appear once in a row
    for (int row = 0; row < SIZE; row++)
      for (int i = 1; i <= 9; i++) {
        int inRow = 0, r1 = 0, c1 = 0;
        for (int col = 0; col < SIZE; col++)
          if (possible[row][col].contains(i)) { inRow++; r1 = row; c1 = col; }
        if (inRow == 1 & hint[r1][c1] == 0) { hint[r1][c1] = i; hints++; }
      }

  // look for numbers that only appear once in a column
    for (int col = 0; col < SIZE; col++)
      for (int i = 1; i <= 9; i++) {
        int inCol = 0, r1 = 0, c1 = 0;
        for (int row = 0; row < SIZE; row++)
          if (possible[row][col].contains(i)) { inCol++; r1 = row; c1 = col; }
        if (inCol == 1 & hint[r1][c1] == 0) { 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]);
        for (int i = 1; i <= 9; i++)
          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)) {
                  keep = i; inBox++; r1 = 3*row + r; c1 = 3*col + c;
                }
            if (inBox == 1 && hint[r1][c1] == 0) { hint[r1][c1] = keep; hints++; }
          }
      }
  } // predictor

  static void fixAndUpdate() {
  // Incorporates the effects of the 3 x 3 submatrices and
  // displays the matrix of possible values and the matrix of already known values

    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");
  IO.writeLine("        0   1   2     3   4   5     6   7   8   ");
  IO.writeLine("     |=============+=============+=============|");
  for (row = 0; row < SIZE; row++) {
    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 assigned\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");
} // fixAndUpdate

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 includes it in the relevant submatrix
  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();
}

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] = all.copy();

  for (row = 0; row <= 2; row++)            // initialize 3 x 3 matrices
    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);
          else {                              // we blew it
            IO.writeLine("Impossible data (" +  n + ") - line " + row + " column " + col);
            System.exit(1);
           }
        }

      fixAndUpdate();                         // display initial matrices

      do {                                    // one move at a time
        if (hints != 0 && automatic) {        // possibly incorporate hints
          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]);
          fixAndUpdate();
        }
        else {                                // human move
          IO.writeLine("Your move - row [0..8] col [0..8] value [1..9] (0 to give up)? ");
          row = IO.readInt(); col = IO.readInt(); n = IO.readInt();
          if (n == 0) System.exit(0);
          if (range.contains(row) && range.contains(col) && possible[row][col].contains(n)) {
                                              // check that it is available
            claim(row, col, n);
            fixAndUpdate();
          }
          else IO.writeLine("******* Impossible");
        }
      } while (count != 81 && choicesRemain);
      if (!choicesRemain) IO.writeLine("\nNo more moves possible");

    }  // main

}  // Sud5
```