Computer Science 301 - 2016

Programming Language Translation

Practical 1, Week beginning 18 July 2016 - Solutions

The submissions received were very varied in quality. There was some innovative work handed in, but there was evidence that people had missed several important points. You can find complete C# source versions of the program solutions in the solution kit `PRAC1A.ZIP` on the server.

Some general comments:

(a) You should *always* put your names and a brief description of the program into your 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, scanners, 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 remember to use the `LPRINT` facility for producing source listings economically. In later practicals the listings will get very wide, and they are hard to read if they wrap round.

Tasks 2 to 6 - The Sieve of Eratosthenes

The Pascal compilers use 16-bit `INTEGER` arithmetic (values from -32768 .. 32767), although they allow very large array sizes, as arrays can also be indexed in some compilers by so-called `LONGINT` (32 bit) variables. And, in fact (probably comes as a surprise to you C-language types), Pascal also allows arrays to have negative indices, so that one can declare, for example

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

However, an array indexed by an INTEGER variable cannot access an element whose subscript is greater than 32767.

Although the Sieve size in the supplied code was apparently "large enough", the Sieve algorithm as supplied could and did easily collapse when applied to a search for large primes, using variables of 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` will eventually become larger than 32767, but if the overflow is not trapped it means that the sum appears to go negative (think back to your CSC 201 course). If you set N to be 20000 this happens for the first time after detecting the prime number 16411, so that the maximum effective sieve algorithm with the code above seems to be limited to primes from 2 to 16411. If you set N to be 32000 it happens for the first time after detecting the prime number 863: in due course the multiple 38*863 is "crossed off" and `K` tries to advance to 32794.

We can extend the range of the algorithm by a trick which I did not really expect you to discover, but which is worth pointing out. Simply replace the above code by something which at first looks ridiculous:

```
K := I (* now cross out multiples of I *);
REPEAT
  Uncrossed[K] := FALSE; K := K + I
UNTIL ((K > N) OR (K < 0))
```

To get this to work you have to set a compiler directive to switch range checking off (using a command line parameter, or inserting a pragma/directive something like {$R-} at the start of the source code). Some students might have got intrigued by all this and probed further (well done). If you try interesting things like "turn off the range checks" on the program as originally supplied, the algorithms appear to allow you to generate higher prime numbers. Trouble is, they might not do it properly, and you find that for some "bigger" values of `Max` you actually seem to find fewer prime numbers generated.

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.

Come on - there must be a better way (there always is). How can the algorithm be changed so that range checks can be left enabled and the system find large primes without bombing?

The C, C++, C# and Parva compilers use 32 bit integers, and thus don't seem to have this problem (or at least, it is much harder to reproduce), but, of course, the amount of real memory available to them may be limited. And how many people noted that the C and C++ source code had declared the size of the array incorrectly?

Executable sizes and limits (as obtained on my machine running Win-7, 32 bit version)

	Empty	Fibo	Sieve	Maximum prime < 32 000 (without trick)	Number of primes < 20 000
Free Pascal FPC (.EXE files) Win7	27716	30276	31300	16411	breaks
Free Pascal FPC (.EXE files) Win 8	57334	61487	62797	16411	breaks
Free Pascal FPC "stripped" Win8	36352 save 20992	39424 save 22063	40448 save 22349	16411	breaks
Turbo Pascal TP6 (.EXE files)	1472	2640	3248	16411	breaks
Turbo Pascal TP60 (.EXE files)	1472	2640	3136	16411	2199 xxx
Turbo Pascal 1 (.COM files)	11386	11601	11971	16411	2199 xxx
C# (.EXE files)	33280	33792	33792	31991	2262
Parva	N/A	N/A	N/A	31991	2262
BCC .C files (.EXE files)	52224	66048	66560	31991	2262
BCC .CPP files (.EXE files)	47104	148480	149504	31991	2262
CL .C files (.EXE files)	21504	34816	34816	breaks	2262
CL .CPP files (.EXE files)	21504	50176	50688	breaks	2262
SieveSet1.exe	33792	N/A	N/A	16411	2262
SieveSet2.exe	33792	N/A	N/A	16411	2262
HanoiIterp2c.exe	34304	N/A	N/A	N/A	N/A
HanoiRecp2c.exe	33792	N/A	N/A	N/A	N/A

The "executables" produced from the C# compiler are not true executables in the same sense as those produced by the C, C++ and Pascal compilers, as the code in them still has to be "jitted" into its final form.

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, although this runs slowly when the checks are present.

The Borland 5.5 and WatCom C/C++ compilers are designed for 32 bit integers and 32-bit operating systems, 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. 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.

Task 6 - 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 should really have been transformed to achieve this. However, even if this is not done it "works". Why?

```
// Sieve of Eratosthenes for finding primes 2 <= n <= Max (Parva version)
// P.D. Terry,  Rhodes University, 2016

void Main() {
  const Max = 32000;
  bool[] uncrossed = new bool[Max];          // the sieve
  int i, n, k, it, iterations, primes = 0;   // counters
  read("How many iterations? ", iterations);
  bool display = iterations == 1;
  read("Supply largest number to be tested ", n);
  if (n > Max) {
    write("n too large, sorry");
    return;
  }
  write("Prime numbers between 2 and " , n, "\n");
  write("-----------------------------------\n");
  it = 1;
  while (it <= iterations) {
    primes = 0;
    i = 2;
    while (i <= n) {                         // clear sieve
      uncrossed[i] = true;
      i = i + 1;
    }
    i = 2;
    while (i <= n) {                         // the passes over the sieve
      if (uncrossed[i]) {
        if (display && (primes - (primes/8)*8 == 0))
          write("\n");                       // ensure line not too long
        primes = primes + 1;
        if (display) write(i, "\t");
        k = i;                               // now cross out multiples of i
        uncrossed[k] = false;
        k = k + i;
        while (k <= n) {
          uncrossed[k] = false;
          k = k + i;
        }
      }
      i = i + 1;
    }
    it = it + 1;
    if (display) write("\n");
  }
  write(primes, " primes");
} // Main
```

Task 7 - High level translators

Some of what might be perceived as "unreadability" presumably relates to the fact that `Parva2ToCSharp` is obliged to translate the read and write multiple parameter functions into a collection of equivalent IO operations from the supporting library.

It is easy to trip up the process. Consider the silly program and its apparently correct translation:

```
void Main () {
  int b;
  bool a = b > 4;
} // Main
```

```
using Library;

class Wrong {

  static public void Main(string[] args) {
    int b;
    bool a = b > 4;
  } // Main

} // Wrong
```

The C# compiler will detect that the assignment statement is meaningless, as `b` has not been initialized, but a Parva compiler and the `Parva2ToCSharp` converter are not as sophisticated.

You may not have seen the point that using a tool like this would allow you to develop and maintain your programs in Parva and then simply convert them to C# when you want to get them compiled on some other system (perhaps so that they can run quickly). So normally a user of `Parva2ToCSharp` would not read or edit the C# code at all. Because of this it is not necessary for the converted code to incorporate the original comments.

If you had played with the `Voter.pav` examples properly you should have found that if all the ages supplied are below 18 there is an attempted division by zero reported. If the program is translated into C#, the same data will generate a corresponding exception. If one were using `Parva2ToCSharp` as a way of speeding up execution one would perhaps be confused by an error message that did not relate back to the original source. And the C# compiler can warn of a variable that is declared but never used, which the Parva one cannot.

Task 8 - The Towers of Hanoi

Task 9 - How fast/slow are various implementations?

Some times (seconds) taken to execute the various programs are shown below, as measured on my computer, which runs Win7-32 (and so could get proper timings for the 16 bit systems as well). Times measured in the lab might have been rather different.

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. Some of 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. There was a script `timer.bat` in the kit which some people tried using to overcome these problems, but it won't work under DosBox

(d) There are clearly some anomalies here. There is something very odd about programs compiled with range checks using Turbo Pascal 6.0, and I have no idea what it can be. However, the general effects are apparent - modern compilers make better use of the large opcode sets on modern processors and produce faster programs suited to the operating systems on those machines. The experiments with the Queens programs show that using global variables is marginally faster then passing parameters (nobody commented on this as I recall, which was a pity). Pascal programs running without range checks execute faster than the same programs compiled to include range checks.

(e) The times taken by the 16-bit systems running under the DOSBox emulator in the lab probably showed 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 in the future if I get the chance.

Timings

	Sieve Number of iterations 10000	Fibo Iterations	Fib4 Iterations	Hanoi Iter Iterations	Hanoi Rec Iterations
	Maximum number tested 15000	Upper limit 40	Upper limit	Number of disks	Number of
Free Pascal FPC (.EXE files)	2.85 ($R+) 2.45 ($R-)	2.15 / 1.51	N/A	N/A	N/A
Turbo Pascal TP6 (.EXE files)	27.25 ($R+) something odd!	> 40	N/A	N/A	N/A
Turbo Pascal TP60 (.EXE files)	1.75 ($R-)	> 40	N/A	N/A	N/A
Turbo Pascal 1 (.COM files)	3.09 ($R+) 2.67 ($R-)		N/A	N/A	N/A
C# (.EXE files)	0.80	1.34	0.02	N/A	N/A
Parva (standard)	33 for 1000 (250 for 10000)	>40	< 0.01		
Parva (optimized)	24 for 1000				
Parva2ToCSharp (.EXE files)	0.78	1.54	0.02		
BCC .C files (.EXE files)	0.67	1.30	N/A	N/A	N/A
BCC .CPP files (.EXE files)	0.69	1.15	N/A	N/A	N/A
CL .C files (.EXE files)	0.62	6.41	N/A	N/A	N/A
CL .CPP files (.EXE files)	0.59	6.32	N/A	N/A	N/A

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 12 - 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, July 2016  ( 2016/07/2416 )

using Library;
using System;

class Sud5 {

  const  int SIZE = 9;
```

```
  static int count = 0, hints;                  // count of the numbers of moves that one can make based on the analy
  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)
    all   = new IntSet(1, 2, 3, 4, 5, 6, 7, 8, 9); // a fully populated set of permissible numbers in a cell prior to
  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
    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 [,]                                // the solution board - populated as numbers are selected from the
    known = new int[9, 9];                      // possible sets for each of its 9 x 9 elements
    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 bool 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.Environment.Exit(1);
    }

    InFile data = new InFile(args[0]);
    if (data.OpenError()) {
      IO.WriteLine("cannot open " + args[0]);
      System.Environment.Exit(1);
    }

    bool automatic = args.Length == 2 && args[1].ToUpper().Equals("-A");

    for (row = 0; row < SIZE; row++)            // initialize the possible matrix
      for (col = 0; col < SIZE; col++)
        possible[row, col] = (IntSet) all.Clone();
                                                // 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.Environment.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.Environment.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("     |=============+=============+=============|");

    // a bit messy, to get neatly formatted output

    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 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
  // 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++)                          // tedious sweeping across to find which of 1..9 it w
            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++) {                            // tedious sweeping across to find which of 1..9 it w
        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++) {                            // tedious sweeping across to find which of 1..9 it w
        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++)                            // bit tedious sweeping across to find which of 1..9
          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
```