Computer Science 301 - 2017

Programming Language Translation

Practical 1, Week beginning 17 July 2017 - 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 - The Sieve of Eratosthenes in Pascal

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, or outside the range specified in the declaration.

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. On the next pass over the loop the now-negative `K` will not be valid as an array index.

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 may 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?

Task 3 - 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, 2017

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
```

The C, C++, C# and Parva compilers use 32 bit integers, and thus don't seem to have overflow problems (or at

least, they are 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? The Watcom version of the incorrect program crashed out, although the Borland compiled version struggled through.

Task 5 - How much code do various compilers generate?

		Empty	Sieve	NPrimes	Fibonacci	Queens1	Queens2	Queens3
FPC Pascal Windows 7-32	EXE	27716	31300	breaks	30276	31300	31300	29252
FPC Pascal Windows 10-64	EXE	57334	62797	breaks	61487	63572	63560	60292
FPO Pascal Optimised	EXE	57334	62157	breaks	61359	62932	62920	59652
FPS Pascal Stripped	EXE	36352	39936	breaks	39424	40448	40448	37888
Turbo Pascal 6 (VDosPlus/DosBox) TP6	EXE	1472	3248	breaks	2640	3808	3632	3472
Turbo Pascal 6 (VDosPlus/DosBox) TP60	EXE	1472	3136	2199 x	2640	3472	3328	3184
Turbo Pascal 3 (VDosPlus/DosBox)	COM	11386	11971	2199 x	11601	12285	12154	12134
Parva	PVM	11	280	2262	110	502	469	479
Parva -O optimize	PVM	11	231	2262	96	403	381	387
C#	EXE	35328	35840	2262	35328	36352	36352	35840
Parva2toC#	EXE	35328	35840	2262	35328	36352	36352	35840
Borland C	EXE	52224	66560	2262	66048			
Borland C++	EXE	47104	149504		148480			
Watcom C	EXE	21504	34816		34816			
Watcom C++	EXE	21504	50688		50176			

Note that the source programs in each case are pretty well equivalent - fairly close manual translations of one another.

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 - 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 will warn if a variable is declared but never used, which the Parva one cannot.

Task 7 and 8 - How efficient is the code generated by various language implementations?

Some times (seconds) taken to execute the various programs are shown below, as measured on a lab computer. I also have one that runs Win7-32 (and so could get proper timings for the 16 bit systems as well).

We may 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. The script `timer.bat` in the kit which some people tried using overcomes these problems for the EXE programs running under Windows, but it won't work under DosBox or VDosPlus.

(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 showed that the solution with global variables isslightly faster than the one with parameters.

(e) The times taken by the 16-bit systems running under the DOS emulators showed quite clearly the adverse effects of emulation. There must be better ways of running old number-crunchers on the latest operating systems, and tools like VM-Ware will be investigated further in the future if I get the chance.

	Iterations & Limit	Sieve	Queens1	Queens2	Queens3	
Iterations			10000 WIN 1000 PAV	1000 DOS 1000 PAV		
Limit			8		13	
FPC Pascal EXE Windows 10-64	4000 15000	0.77	1.46	1.42	0.33	
FPO Pascal EXE Windows 10-64	4000 15000	0.64	1.33	1.20	0.33	
Turbo Pascal 6 EXE Windows 7-32 TP6	4000 15000	10.9	20.28	16.96	7.33	
Turbo Pascal 6 EXE Windows 7-32 TP60	4000 15000	0.72	3.89	1.97	7.33	
Turbo Pascal 3 COM Windows 7-32	4000 15000	1.59	6.90	2.82	3.18	
Turbo Pascal 6 EXE VDosPlus TP6	400 15000	6.06	10.35	8.89	36.48	
Turbo Pascal 6 EXE VDosPlus TP60	400 15000	2.24	4.44	3.04	15.0	
Turbo Pascal 3 COM VDosPlus	400 15000	2.67	8.19	4.69	32.48	
Turbo Pascal 6 EXE DosBox TP6	400 15000	7.5	12.5	11.22	45.0	
Turbo Pascal 6 EXE DosBox TP60	400 15000	2.22	4.41	3.2	15.4	
Turbo Pascal 3 COM DosBox	400 15000	2.75	9.23	5.3	39.0	
Parva PVM	400 15000	12.56	15.12	14.94	10.2	check
Parva -O optimize PVM	400 15000	9.91	11.85	11.17	7.84	check
C# EXE Windows 10-64	4000 15000	0.33	0.99	1.11	0.42	
Borland C EXE Windows 10-64	4000 15000	0.25				
Borland C++ EXE Windows 10-64	4000 15000	0.27				
Watcom C EXE Windows 10-64	4000 15000	0.27				
Watcom C++ EXE Windows 10-64	4000 15000	0.23				

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

Task 10 One way of speeding up tedious recursion

A number of people missed the point badly. Their solutions simply became equivalent to a "fast iterative" method, which worked simply because the *while* loop on the main routine worked upwards!

The sort of solution I was looking for is as follows:

```
// Print a table of Fibonacci numbers using (fast) recursive definition
// and memoisation
// P.D. Terry,  Rhodes University, 2017
```

```
int[] fibmem = new int[4000];

int fib(int m) {
// Compute m-th term in Fibonacci series 0,1,1,2 ...
  if (m == 0) return 0;
  if (m == 1) return 1;
  if (fibmem[m] == 0)
    fibmem[m] = fib(m-1) + fib(m-2);       // store the recursion value
  return fibmem[m];
} // fib

void main() {
  int limit;
  read("Supply upper limit ", limit);
  int i = 0;
  while (i <= limit) {
    fibmem[i] = 0;
    i = i + 1;
  }
  i = 0;
  while (i <= limit) {
    write(i, "\t", fib(i), "\n");
    i = i + 1;
  } // while
} // main
```

Even if the function were to be called for the first time with a large value of the arguments, as in this variation

```
  i = limit;
  while (i >= 0) {
    write(i, "\t", fib(i), "\n");
    i = i - 1;
  } // while
```

it would force a recursive chain that would rapidly fill the entire array (puzzle this one out if it is not obvious) and further calls for smaller arguments could then pick the values stored in the array.

The technique is called "memoization". I was pleased to see that some students had come across this rather strange name.

Task 11 - Creative programming - Goldbach's conjecture

Where they had bothered to try this one, several groups had grasped the concept of setting up a set to contain the prime numbers. Many had not realised that one could use a set in place of the boolean array in order to apply the Sieve algorithm in the first place, so have a look at the code below and see how easy this is to do, and how the the sieve is constructed from a simple adaptation of the code given to you earlier. There is, of course, no need for the `primes` method to do any output, or even to count how many prime numbers are added to the set. The `IntSet` class, like the `List` class in C# is "elastic", expanding automagically when necessary, unlike a simple array as used in the original code.

There was some very tortuous and confused code submitted thereafter for the very simple task of trying to find whether an even number could be expressed as the sum of two of those primes. You only need one loop for each attempt - if N is to be expressed as the sum of two numbers A and B, then there is no need to try out all possible combinations of A and B (since B must = N - A). Additionally, we need only test values for A from 2 to at most N/2 (think about it!). And we can make the system even more efficient if we use *while* loops rather than fixed *for* loops, and stop the loops early when it is clear that there is no need to continue.

So one way of programming this exercise would be as below.

```
// Sieve of Eratosthenes (in a set) for testing Goldbach's conjecture
// P.D. Terry,  Rhodes University, 2017

import library.*;

class Goldbach {

  public static void main(String [] args) {
    int limit = IO.readInt("Supply largest number to be tested ");
    IntSet primeSet = primes(limit);
    boolean conjecture = true;        // optimistic
    int test = 4;
    while (conjecture && test <= limit) {
      boolean found = false;          // try to find the pair
```

```
      int i = 2;
      while (i <= test / 2 && !found) {
        if (primeSet.contains(i) && primeSet.contains(test - i)) { // short-circuit helps too!
          found = true;
          IO.writeLine(" " + test + "\t" + i + "\t" + (test - i));
        }
        else i++;
      }
      if (!found) {
        IO.writeLine("conjecture fails for ", test);
        conjecture = false;
      }
      test = test + 2;                // move on to next even number
    }
    IO.writeLine("Conjecture seems to be " + conjecture);   // final result of test
  } // main

  static IntSet primes(int max)  {
  // Returns the set of prime numbers smaller than max
    IntSet primeSet = new IntSet();   // the prime numbers
    IntSet crossed = new IntSet();    // the sieve
    for (int i = 2; i <= max; i++) {  // the passes over the sieve
      if (!crossed.contains(i)) {
        primeSet.incl(i);
        int k = i;                    // now cross out multiples of i
        do {
          crossed.incl(k);
          k += i;
        } while (k <= max && k > 0);
      }
    }
    return primeSet;
  } // primes

} // Goldbach
```

Terry Theorem 1: You can improve on almost any program if you think about it. The code above still does about twice as much work as it needs to do. Why - and how could you improve it by a very simple modification?

Task 12 - Something more creative - "Pig Latin" in C#

A naïve solution for this can be effected quite easily. The code below does this for data read word by word - rather than line by line - from a data file specified as a program "arg" and storing the result in another text files specified by another "arg". There was no need to tokenize, simply to use the `ReadWord` method in my library).

I do not recall many submissions that used the file opening methods correctly, unfortunately

```
// Convert an English text to "Pig Latin"
// Words may only contain letters
// P.D. Terry, Rhodes University, 2017

using Library;
using System;
using System.Text;

class ToLatin1 {

  static string Convert(string s) {
  // Convert s to Pig Latin - move first letter to end and then append "ay"
  // For example, "program" is returned as "rogrampay"
  // Words may only contain letters
    if (s.Length > 0) s = s.Substring(1) + s[0] + "ay";
    return s;
  } // Convert

  public static void Main(string[] args) {
  // first check that command line arguments have been supplied
  // attempt to open data file    using file names from "args" ++++++++++++++++++++++++++
  // attempt to open results file and check that the files open correctly ++++++++++++++++
  // all this as in SampleIO.cs - see full solution for details

  // read and process data file
    while (true) {
      string word = data.ReadWord();
      if (data.NoMoreData()) break;
      results.Write(Convert(word));
      if (data.EOL()) results.WriteLine(); else results.Write(' ');
    }

  // close results file safely
    results.Close();
  } // Main

} // ToLatin1
```

The corresponding program for decoding is essentially the same, with the `Convert` method replaced by a `Deconvert` method:

```
static string Deconvert(string s) {
// Convert a Pig Latin word to English - check for the "ay" at the end
// In this version the word proper is assumed to contain only letters
  int ay = s.LastIndexOf("ay");
  if (ay >= 1) s = s[ay-1] + s.Substring(0, ay - 1);
  return s;
} // Deconvert
```

While several people submitted solutions on these lines, few thought to check that the conversion or deconversion would actually be possible. Students, of course, are idealists and fondly believe that data will always be perfect and that nothing can ever go wrong - but it most certainly can, and by this stage of your careers you should be thinking all the time of how to write really reliable code.

A few submissions, I was pleased to say, had realized that "words" might not always be composed only of letters, but might be numbers, or be preceded or followed by punctuation marks:

> She said: "I have found 100 mistakes in your wonderful textbook".

Situations like this are a bit harder to handle. Here are some possibilities that make use of the `StringBuilder` class, which you should learn about, as it is very useful for manipulating strings dynamically. I have deliberately left this code badly commented, just to drive home the point that reading someone else's uncommented code often takes considerable effort.

```
static string Convert(string s) {
// Convert s to Pig Latin - move first letter to end and then append "ay"
// For example, "program" is returned as "rogrampay"
// Words may start and end with non-letters which remain there
// For example "1234" is returned as 1234; "Hello!!" as "elloHay!!"
  StringBuilder sb = new StringBuilder();
  int i = 0, sl = s.Length;
  while (i < sl && !Char.IsLetter(s[i])) {
    sb.Append(s[i]); i++;
  }
  if (i < sl) {
    char first = s[i];
    i++;
    while (i < sl && Char.IsLetter(s[i])) {
      sb.Append(s[i]); i++;
    }
    sb.Append(first);
    sb.Append("ay");
    while (i < sl) {
      sb.Append(s[i]); i++;
    }
  }
  return sb.ToString();
} // Convert

static string Deconvert(string s) {
// Convert a Pig Latin word to English - check for the "ay" at the end
// In this version non-letters can precede and follow the word proper
  int ay = s.LastIndexOf("ay");
  if (ay < 1) return s;
  StringBuilder sb = new StringBuilder();
  int i = 0;
  while (i < ay - 1 && !Char.IsLetter(s[i])) {
    sb.Append(s[i]); i++;
  }
  sb.Append(s[ay-1]);
  sb.Append(s.Substring(i, ay - i - 1));
  int L = s.Length;
  if (L - ay - 2 > 0) sb.Append(s.Substring(ay + 2, L - ay - 2));
  return sb.ToString();
} // Deconvert
```

Of course the situation is really even more complicated - there might be words with interior punctuation:

> He replied, sadly, "That's very clever of you to find so many, my dear Gambol Hedge-Bette".

but the refinements needed to handle this are left as an exercise.