Computer Science 301 - 2014

Programming Language Translation

Practical 1, Week beginning 24 March 2014 - Solutions

The submissions received were very varied, but on the whole of rather poor 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 `PRAC1A.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 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 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 probably 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 was imposed - an array indexed by an INTEGER variable could not get 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 as supplied 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 seems to be limited to primes from 1 to 16411.

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

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

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 is 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.

	Sieve Code Size	Fibo Code Size	Empty Code Size		
Turbo Pascal 1	11 751	11 601	11 386	Max prime ? 16411	.COM file sizes
Turbo Pascal 6	3 184	2 640	1 472	Max prime ? 16411	.EXE file sizes
Optimized Turbo Pascal 6	3 104	2 640	1 472	Max prime ? 16411	.EXE file sizes
Free Pascal	34 612	33 588	30 516	Max Prime ? 16411	.EXE file sizes
Optimized Free Pascal	34 100	33 588	30 516	Max prime ? 16411	.EXE file sizes
Borland C	66 560	66 048	52 224		.EXE file sizes
Borland C++	149 504	148 480	47 104		.EXE file sizes
Watcom C	34 816	34 816	21 504		.EXE file sizes
Watcom C++	50 688	50 176	21 504		.EXE file sizes
C#	32 768	32 256	31 744		.EXE file sizes
Parva	N/A	N/A	N/A	Sieve limit 49000	

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.

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?

```
void main() {
// Sieve of Eratosthenes for finding primes 2 <= n <= 49000 (Parva version)
// P.D. Terry,  Rhodes University, 2014
  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)      // this is equivalent to primes % 8 - not many knew this!!
          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
```

Task 8 - High level translators

Some of what may perceived as "unreadability" presumably relates to the fact that `Parva2ToJava` is obliged to translate the read and write multiple parameter functions into a collection of equivalent IO operations from the supporting library. Some people commented that `Parva2ToJava` seemed capable of generating Java code that could not be compiled, but had not noticed that in this case there had been error messages during the translation into Java. Of course, what should have happened in this case was that the bad Java code should have been deleted before the program ceased execution, and this has now been done. (Did I not tell you that the best way to test a compiler is to let a lot of students loose trying to use it?)

Not everyone may have seen the point of that using a tool like this would allow you to develop and maintain your programs in Parva and then simply convert them to Java when you want to get them compiled on some other machine (berhaps so that they can run quickly!). So normally a user of `Parva2ToJava` would not read or edit the Java code at all. Because of this it is not necessary for the converted code to incorporate the original comments.

Task 10 - 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). There was a script `timer.bat` in the kit which some people tried using to overcome these problems, which really requires slight further modification of the code being timed so as to eliminate the interactive input phase.

(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		
C	3.8	1.54	2.2	1.2	1.1	
C++	4.1	1.55	2.1	1.1	1.0	
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 in the future if I get the chance.

Fibo: Upper limit 40							
	1GHz LT	2GHz LT	3GHz PC	2GHz i7	3GHz I5	3GHz I5	
	XP 32	XP 32	XP 32	XP 32	Win7 64	DosBox	
Turbo Pascal 1.0		7.60		4.1		323	Ignore overflow
Turbo Pascal 6.0		31.00		17.7		275	
Turbo Pascal 6.0 (opt)		31.00		17.7		275	
Free Pascal	12.7	4.54	4.2	2.6	2.4		
Free Pascal (optimized)	12.3	4.33	4.1	2.5	2.1		
C	12.3	5.60	3.9	2.5	1.7		
C++	12.2	5.44	4.0	2.3	1.6		
C#	9.8	5.00	5.1	2.3	1.6		
Java (with JIT)	9.3	2.56	2.7	1.3	1.2		
Java -Xint (interpret)	92.6	26.00	48.2	17.4	13.3		
Parva	692	317	277.0	122.0			
Parva (optimized)		257	225.0	100.0			

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 13 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! Next time I use this exercise I shall try to remember to suggest that it work downwards.

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, 2014

  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 14 Creative programming - How long will my journey take?

The main point of this exercise was to give some exposure to the I/O libraries - especially the input routines - and the `ArrayList` class, which we shall use repeatedly later on as a simple container class for building up symbol tables and the like. For ease of use, the little `BusStop` class in the suggested code below has exposed its data members as "public", thus eliminating the need for "getter" methods.

```
  // Set up a poster of travel times between any two stops on a bus route
  //
  // Data is of the form
  //
  //   College 8 Hamilton 5 Oakdene 12 Gino's 25 Mews 9 Union 12 Steers 17 Athies
  //
  // P.D. Terry, Rhodes University, 2014

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

  class BusStop {
    public String name;
    public int toNext;

    public BusStop(String name, int toNext) {
      this.name = name;
      this.toNext = toNext;
    } // constructor

  } // BusStop

  class Poster {

    public static void main(String[] args) {
      ArrayList<BusStop> list = new ArrayList<BusStop>();
      int i, j, toNext;

      //              Construct list of bus stop and travel times

      do {
        String stationName = IO.readWord();          // easy way to read a simple word
        toNext = IO.readInt();                       // easy way to read and decode an integer
        if (IO.noMoreData()) toNext = 0;             // easy way to detect when data is exhausted
        list.add(new BusStop(stationName, toNext));
      } while (toNext > 0);

      //              Construct symmetric matrix of accumulated travel times

      int[][] matrix = new int[list.size() + 1] [list.size() + 1];
      for (i = 0; i <= list.size(); i++) {
        matrix[i][i] = 0;
        for (j = i + 1; j < list.size() + 1; j++)
          matrix[j][i] = matrix[i][j] = matrix[i][j-1] + list.get(j-1).toNext;
      }

      //              Output bus stop names and matrix

      IO.write(" ", 12);
      for (i = 0; i < list.size(); i++)
        IO.write(list.get(i).name, 10);              // easy way to use a fixed size output field
      IO.writeLine();
      IO.writeLine();
      for (i = 0; i < list.size(); i++) {
        IO.write(list.get(i).name, -10);             // -10  : left justified for neatness
        for (j = 0; j < list.size(); j++)
          IO.write(matrix[i][j], 10);                // +10  : right justified for neatness
        IO.writeLine();
      }

    } // main

  } // Poster
```

Some submissions made use of the `InFile` and `OutFile` libraries, rather than relying on redirection from the command prompt. Just for fun, here is the C# version of such an implementation from which you will see how close C# and Java are. (I happen to think array/matrix subscripting is easier in C#.)

Frankly, some of the submissions were bizarre. The I/O input routines in the Terry library are very versatile. There is no need to read everything as strings, or as one long line, and then convert to and from strings and integers all over the show. I guess you do this because in spite of my best efforts to persuade colleagues otherwise, you were persuaded by said colleagues to use the `Scanner` class …

```
// Set up a poster of travel times between any two stops on a bus route
// Data is of the form
//   College 8 Hamilton 5 Oakdene 12 Gino's 25 Mews 9 Union 12 Steers 17 Athies
// This version uses InFile and OutFile libraries rather than redirecting I/O with the IO library
// P.D. Terry, Rhodes University, 2014

using System;
using Library;
using System.Collections.Generic;

class BusStop {
  public string name;
  public int toNext;

  public BusStop(string name, int toNext) {
    this.name = name;
    this.toNext = toNext;
  } // constructor

} // BusStop

class Poster {

  public static void Main(string[] args) {
    List<BusStop> list = new List<BusStop>();
    int i, j, toNext;

    //               Command line argument processing
    //               Check that arguments have been supplied
    if (args.Length != 2) {
      Console.WriteLine("missing args");
      System.Environment.Exit(1);
    }

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

    //               Attempt to open results file
    OutFile results = new OutFile(args[1]);
    if (results.OpenError()) {
      Console.WriteLine("cannot open " + args[1]);
      System.Environment.Exit(1);
    }

    //               Construct list of bus stop and travel times
    do {
      string stationName = data.ReadWord();
      toNext = data.ReadInt();
      if (data.NoMoreData()) toNext = 0;
      list.Add(new BusStop(stationName, toNext));
    } while (toNext > 0);

    //               Construct symmetric matrix of accumulated travel times
    int[,] matrix = new int[list.Count + 1, list.Count + 1];
    for (i = 0; i <= list.Count; i++) {
      matrix[i, i] = 0;
      for (j = i + 1; j < list.Count + 1; j++)
        matrix[j, i] = matrix[i, j] = matrix[i, j-1] + list[j-1].toNext;
    }

    //               Output bus stop names and matrix
    results.Write(" ", 12);
    for (i = 0; i < list.Count; i++) results.Write(list[i].name, 10);
    results.WriteLine();
    results.WriteLine();
    for (i = 0; i < list.Count; i++) {
      results.Write(list[i].name, -10);
      for (j = 0; j < list.Count; j++) results.Write(matrix[i, j], 10);
      results.WriteLine();
    }
    results.Close();
  } // Main

} // Poster
```

Task 15 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 `ArrayList` (Java) or `List` class (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, 2014

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?