

Computer Science 301 - 2009

Programming Language Translation

Practical for Week 19, beginning 31 August 2009 - 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) A large number of 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.

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 800GB of disk space, and if they don't they should go and buy more" philosophy.

The "executable" 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.

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 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 | Queens Code Size | Empty Code Size | |
|------------------------|-----------------|------------------|-----------------|-------------------|
| Free Pascal | 31 744 | 32 256 | 28 672 | Sieve Limit 16410 |
| Optimized Free Pascal | 31 744 | 32 256 | 28 672 | Sieve Limit 16410 |
| Modula-2 | 18 609 | 18 774 | 11 946 | Sieve Limit 32770 |
| Optimized M-2 | 18 549 | 18 492 | 11 946 | Sieve Limit 32770 |
| Borland c | 66 560 | | 52 224 | |
| Borland c++ | 149 504 | | 47 104 | |
| Watcom c | 37 376 | | 27 648 | |
| Watcom c++ | 51 712 | | 22 528 | |
| C# | 40 960 | 40 960 | 40 960 | |
| Parva | N/A | N/A | N/A | Sieve Limit 49000 |
| Modula-2 via Borland c | 62 976 | 63 488 | 62 464 | |

The Sieve in Parva

The corrected code is very simple. A few groups got it badly wrong, with braces in the wrong places.

```
void main() {
// Sieve of Eratosthenes for finding primes 2 <= n <= 49000 (Parva version)
// P.D. Terry, Rhodes University, 2009
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) {
        Uncrossed[i-2] = true;
        i = i + 1;
    }
    i = 2;
    while (i <= n) {
        if (Uncrossed[i-2]) {
            if (primes - (primes/8)*8 == 0)
                write("\n");
            primes = primes + 1;
            write(i, "\t");
            k = 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 - The N Queens problem

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

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

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

Task 9 - High level translators

Several students complained that the C code generated by the X2C system was "unreadable", and "not what we would have written ourselves". There can be no dispute with the second of those arguments, but if you take a careful look at the generated C code, it is verbose, rather than unreadable, because long identifier names have been used. This is actually not such a bad thing - at least one can tell the "origin" of any identifier, since the original module in which it was declared is incorporated into its name, and this is 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 not one some of you will ever 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 Xtacy would not read or edit the C code at all.

Task 10 - How fast/slow are various implementations?

The times (seconds) taken to execute the various programs are shown below (figures in brackets are for runs on my late lamented old 1.04GHz laptop, others on 3.00 GHz machines). In all cases the systems ran on Windows XP.

| | | | | | | |
|--------------------------|----------------------|-------|---------|-------|--------|-------|
| Sieve: Iterations 10 000 | Size of sieve 16 000 | | | | | |
| Queens: Iterations 100 | Board Size 11 | | | | | |
| | Sieve | | Queens | | | |
| | | | Queens1 | | | |
| Free Pascal | (9.2) | 4.4 | (5.5) | 3.2 | (5.2) | 3.1 |
| Free Pascal (optimized) | (8.6) | 4.2 | (5.7) | 3.1 | (5.0) | 2.9 |
| Modula-2 | (5.7) | 2.4 | (9.2) | 7.5 | (7.4) | 6.6 |
| Modula-2 (optimized) | (2.3) | 1.8 | (8.0) | 5.8 | (5.6) | 4.3 |
| C | (3.8) | 2.2 | | | | |
| C++ | (4.1) | 2.1 | | | | |
| Modula-2 (via BCC) | (25.2) | 6.3 | (11.0) | 4.6 | (10.7) | 4.3 |
| C# | (3.7) | 3.2 | (3.8) | 2.9 | (4.0) | 2.8 |
| Java (with JIT) | (6.6) | 3.0 | (5.4) | 2.6 | (5.3) | 2.8 |
| Java -Xint (interpret) | (74.3) | 46.6 | (33.3) | 23.6 | (37.8) | 27.2 |
| Parva | (1080) | 448.0 | (434) | 178.0 | (425) | 186.0 |
| Parva (optimized) | (810.0) | 360.0 | (324) | 139.0 | (326) | 139.0 |

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. The output statements were commented out so that all that was really being measured was the time for the algorithms themselves.
- (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.

Task 12 Creative Parva programming - the "hailstone" sequence

A hailstone sequence is easily generated, save for the annoyance of not having the % (modulo) operator and an *else* for the *if* statement.

One simple solution follows

```
/* Solve the hailstone series problem
   P.D. Terry, Rhodes University, 2009 */

int lengthOfSeries(int term) {
// Returns the length of the series that starts at term
  int n = 1, next;
  while (term != 1) {
    n = n + 1;
    if (term / 2 * 2 != term) next = 3 * term + 1;
    if (term / 2 * 2 == term) next = term / 2;
    term = next;
  }
  return n;
}

void main () {
// Finds the smallest initial term so that we get a run of more than limit
// terms in the hailstone series
  int limit;
  read("What is the length to exceed?", limit);
  int start = 0, length = 0;
  while (length < limit) {
    start = start + 1;
    length = lengthOfSeries(start);
  }
  write("Smallest initial number was ", start, " leading to ", length, " terms");
}
```

Here is part of another solution, which uses the property of the *return* statement to compensate for the lack of an *else*:

```
int nextTerm(int term) {
// return next term in a hailstone sequence
  if (term / 2 * 2 != term) return 3 * term + 1;
  return term / 2;
}

int lengthOfSeries(int term) {
// return the length of the series that starts at term
  int n = 1;
  while (term != 1) {
    n = n + 1;
    term = nextTerm(term);
  }
  return n;
}
```

Some people might prefer to think recursively (perhaps a little less neatly than this):

```
int lengthOfSeries(int term) {
// return the length of the series that starts at term
  if (term == 1) return 1;
  if (term / 2 * 2 != term) return 1 + lengthOfSeries(3 * term + 1);
  return 1 + lengthOfSeries(term / 2);
}
```

A recursive solution would be considerably less efficient than an iterative one, as it would involve the overhead of multiple function calls and parameter passing.

Task 12 Creative programming - How long will my journey take?

The main point of this exercise was to give some exposure to 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 `Station` class in the 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 stations on a suburban system
//
// 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, 2009

```

```

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

class Station {
    public String name;
    public int toNext;

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

class Poster {

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

        // Construct list of stations and travel times

        do {
            String stationName = IO.readWord();
            toNext = IO.readInt();
            if (IO.noMoreData()) toNext = 0;
            list.add(new Station(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 station names and matrix

        IO.write(" ", 12);
        for (i = 0; i < list.size(); i++) IO.write(list.get(i).name, 10);
        IO.writeLine();
        IO.writeLine();
        for (i = 0; i < list.size(); i++) {
            IO.write(list.get(i).name, -10);
            for (j = 0; j < list.size(); j++) IO.write(matrix[i][j], 10);
            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.

```

// Set up a poster of travel times between any two stations on a suburban system
//
// 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, 2009
//
// This version uses InFile and outFile libraries rather than redirection I/O with the IO library

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

```

```

class Station {
    public String name;
    public int toNext;

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

class Poster {

    public static void Main(string[] args) {
        List<Station> list = new List<Station>();
        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 stations and travel times

        do {
            string stationName = data.ReadWord();
            toNext = data.ReadInt();
            if (data.NoMoreData()) toNext = 0;
            list.Add(new Station(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 station 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 14 Creative programming - Goldbach's conjecture

Most submissions 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 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.

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.

```
import library.*;

class Goldbach {
// Sieve of Eratosthenes for testing Goldbach's conjecture
// P.D. Terry, Rhodes University, 2009

public static void main(String [] args) {
    int limit = IO.readInt("Supply largest number to be tested ");
    IntSet primeSet = primes(limit);
    boolean Goldbach = true; // optimistic
    int test = 4;
    while (Goldbach && 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)) {
                found = true;
                IO.writeLine(" " + test + "\t" + i + "\t" + (test - i));
            }
            else i++;
        }
        if (!found) {
            IO.writeLine("conjecture fails for ", test);
            Goldbach = false;
        }
        test = test + 2; // move on to next even number
    }
    IO.writeLine("Conjecture seems to be " + Goldbach); // 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);
    }
}
return primeSet;
} // primes
} // Goldbach
```