Computer Science 301 - 2012

Programming Language Translation

Practical for Week 19, beginning 27 August 2012 - Solutions

The submissions received were very varied, but on the whole of rather reasonable quality. There was some innovative work handed in, but there was evidence that people had missed several important points. You can find complete source versions of the program solutions in the solution kit `PRAC19A.ZIP` on the server. This file also contains C# versions of the solutions for people who might be interested.

Some general comments:

(a) You should *always* put your names and a brief description of the program into the source code.

(b) Several submissions had almost no commentary at all, and this is just unacceptable. In particular, supply commentary at the start of each method as to what it sets out to do, and on the significance of the parameters/arguments.

(c) The pracs in this course are deliberately set to extend you, in the hope that you will learn a lot from each one. Their completion requires that you apply yourself steadily throughout the week, and not just on Thursday afternoon and the following Thursday morning!

(d) Some submissions were received that had not made proper use of the `IO`, `InFile` and `OutFile` classes *as you had been told to do*. These library classes are designed to make text based I/O as simple as possible without having to deal with buffered readers, exceptions, string tokenizers and all that stuff that you were probably subjected to in CSC 102, but without 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 obviously 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 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` should really 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 (currently 2012).) 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 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.

Execution times were slightly different, however.

	Sieve Code Size	Fibo Code Size	Empty Code Size		
Turbo Pascal 1	11 751	11 601	11 386	Sieve limit 16410	.COM file sizes
Turbo Pascal 6	2 912	2 640	1 472	Sieve limit 16410	.EXE file sizes
Optimized Turbo Pascal 6	2 848	2 640	2 640	Sieve limit 16410	.EXE file sizes
Free Pascal	34 612	33 588	30 516	Sieve limit 16410	.EXE file sizes
Optimized Free Pascal	34 100	33 588	30 516	Sieve limit 16410	.EXE file sizes
Modula-2	18 609	18 098	11 946	Sieve limit 32770	.EXE file sizes
Optimized M-2	18 549	18 049	11 946	Sieve limit 32770	.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#	32 256	31 744	31 744		.EXE file sizes
Parva	N/A	N/A	N/A	Sieve limit 49000	
Modula-2 via Borland C	62 976	62 464	62 464		.EXE file sizes

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, 2012
  const Max = 49000;
  bool[] Uncrossed = new bool[Max];         // the sieve
  int i, n, k, it, iterations, primes = 0;  // counters
  read("How many iterations? ", iterations);
  read("Supply largest number to be tested ", n);
  if (n > Max) {
    write("n too large, sorry");
    return;
  }
  it = 1;
  while (it <= iterations) {
    primes = 0;
    write("Prime numbers between 2 and " , n, "\n");
    write("-----------------------------------\n");
    i = 2;
    while (i <= n) {                          // clear sieve
      Uncrossed[i-2] = true;
      i = i + 1;
    }
    i = 2;
    while (i <= n) {                          // the passes over the sieve
      if (Uncrossed[i-2]) {
        if (primes - (primes/8)*8 == 0)
          write("\n");                        // ensure line not too long
        primes = primes + 1;
        write(i, "\t");
        k = i;                                // now cross out multiples of i
        Uncrossed[k-2] = false;
        k = k + i;
        while (k <= n) {
          Uncrossed[k-2] = false;
          k = k + i;
        }
      }
      i = i + 1;
    }
    it = it + 1;
    write("\n");
  }
  write(primes, " primes");
} // main
```

Task 8 - High level translators

Several students complained that the C code generated by the X2C system was "unreadable", and "not what we would have written ourselves". There can be no dispute with the second of those arguments, but if you take a careful look at the generated C code, it is verbose, rather than unreadable, because long identifier names have been used. This is actually not such a bad thing - at least one can tell the "origin" of any identifier, since the original module in which it was declared is incorporated into its name, and this is is a very useful trick, both for large programs, and (more especially) when one has to contend with the miserable rules that C has for controlling identifier name spaces. In fact, in the days when I used Modula regularly, I used a convention similar to this of my own accord, and have carried it over into my other coding, as you will see in several places in the book. Some of the other "unreadability" presumably relates to the fact that the X2C system is obliged to translate the `CARDINAL` type to the `unsigned int` type, which is one some of you will never have used - this explains all those funny casts and capital Us that you saw. Some people commented that "maintaining the C code generated would be a nightmare". Well, maybe, but the point of using a tool like this is that you can develop and maintain your programs in Modula-2 and then simply convert them to C when you want to get them compiled on some other machine. So normally a user of X2C would not read or edit the C code at all.

Task 9 - How fast/slow are various implementations?

Some times (seconds) taken to execute the various programs are shown below. for running these systems over several years, computers and operating systems. In earlier times the systems ran Windows XP-32.

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).

(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.

(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 next year.

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		
Modula-2	5.7	2.60	2.4	1.4		82
Modula-2 (optimized)	2.3	1.17	1.8	0.8		35
C	3.8	1.54	2.2	1.2	1.1	
C++	4.1	1.55	2.1	1.1	1.0	
Modula-2 (via C)	25.2	6.04	6.3	1.4	3.5	
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	

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.

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		
Modula-2		21.60		12.6		360.0	Using LONGCARD
Modula-2 (optimized)	23.5	16.47	32.3	12.7		205.0	
C	12.3	5.60	3.9	2.5	1.7		
C++	12.2	5.44	4.0	2.3	1.6		
Modula-2 (via C)	18.9		3.9	2.2	1.5		
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			

Task 11 - 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, 2012 */

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");
}
```

This might be improved further as follows. Note that it is necessary to ensure that only one term is computed on each pass through the loop. Several people did not realize this, while several other had very inelegant ways of achieving it.

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

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;
} // nextTerm

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;
} // lengthOfSeries
```

Some people might have thought recursively (perhaps a little less neatly than this, however):

```
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 - Lucky numbers

Some very complicated solutions were received. It is possible to solve this problem in several ways. One way is to move numbers from the higher parts of the list so as to let them overwrite numbers in the lower part of the list that are no longer relevant. A solution that exploits this idea follows:

```
// Simple test bed program for checking on the generation of lucky numbers
// P.D. Terry, Rhodes University, 2012

int ReduceLength (int[] list, int n, int length) {
// Reduce list of length numbers by removing every n-th number
// Return length of reduced list
  int copied = 0;
  int i = 1;
  while (i <= length) {
    if (i - (i / n) * n != 0) {  // we would remove one if i % n == 0
      list[copied] = list[i-1];
      copied = copied + 1;
    }
    i = i + 1;
  }
  return copied;
}
```

```
int GenerateLuckyNumbers(int[] list, int max) {
// Generate a list of the lucky numbers between 1 and max (inclusive)
// and return the length of the sequence
  int length = max, i = 0;
  while (i < length) {  // generate original fully populated list
    list[i] = i + 1;
    i = i + 1;
  }
  int step = 2;
  while (step <= length) {
    length = ReduceLength(list, step, length);
    step = step + 1;
  }
  return length;
} // GenerateLuckyNumbers

void main() {
  int limit;
  read("Supply limit of numbers to be tested for luck ", limit);
  int[] luckyList = new int[limit];
  int n = GenerateLuckyNumbers(luckyList, limit);
  int i = 0;
  while (i < n) {
    write(luckyList[i], "\t");
    i = i + 1;
  }
} // main
```

A solution that I received some years ago when I used this exercise intrigued me, as at first I thought it must be incorrect. On further investigation it turned out to be correct, but a little clumsy, in that it made far more passes over the list than were needed. A little thought showed that it could be improved quite easily. It's always fun to be shown things one had not thought of earlier, so for the record, here is my version of this submission. This algorithm does not move the numbers around in the array, and so has the disadvantage that one has to keep scanning over the whole of the long array on each pass; in the solution above, the list rather rapidly gets shorter.

```
// Another simple program for generating a list of lucky numbers
// P.D. Terry, Rhodes University, 2012
// Based on an idea by P. Heideman, D. Terry and M. Whitfield, 2003

int mod(int x, int n) {
  return x - (x/n)*n;
} // mod

void main () {
  int limit;
  read("Supply limit of numbers to be tested for luck ", limit);

//     generate an initial sieve
  bool[] lucky = new bool[limit];
  int i = 0;
  while (i < limit) {
    lucky[i] = true;
    i = i + 1;
  }
//     perform passes over the sieve
  int step = 2;
  int remaining = limit;
  while (step <= remaining) {
    i = 0;
    remaining = 1;
    while (i < limit) {
      if ((mod(remaining, step) == 0) && lucky[i]) {
        lucky[i] = false;
        remaining = remaining + 1;
      }
      if (lucky[i]) remaining = remaining + 1;
      i = i + 1;
    }
    step = step + 1;
  }
//     display the surviving lucky numbers
  i = 0;
  while (i < limit) {
    if (lucky[i]) write(i + 1);
    i = i + 1;
  }
} // main
```

Many of the submitted solutions created a new array on each pass, copied elements of the old array to this new array, and then assigned the pointer to the new array to the old pointer. While this works for small lists, in the absence of proper garbage collection (as is the case with Parva at this stage) one rapidly exhausts the heap and the system collapses.

Task 13 - Pig Latin

A naive 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 (no need to tokenize, simply use the `readWord` method in my library).

```
// Convert an English text to "Pig Latin"
// Words may only contain letters
// P.D. Terry, Rhodes University, 2012

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

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.charAt(0) + "ay";
    return s;
  }

  public static void main (String[] args) {
  // first check that commandline arguments have been supplied
  // attempt to open data file
  // attempt to open results file
  // all this as in SampleIO.java - 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();
  }

}
```

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.charAt(ay-1) + s.substring(0, ay - 1);
    return s;
  }
```

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:

> He 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 `StringBuffer` 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!!"
  StringBuffer sb = new StringBuffer();
  int i = 0, sl = s.length();
  while (i < sl && !Character.isLetter(s.charAt(i))) {
    sb.append(s.charAt(i)); i++;
  }
  if (i < sl) {
    char first = s.charAt(i);
    i++;
    while (i < sl && Character.isLetter(s.charAt(i))) {
      sb.append(s.charAt(i)); i++;
    }
    sb.append(first); sb.append("ay");
    while (i < sl) {
      sb.append(s.charAt(i)); i++;
    }
  }
  return sb.toString();
}

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 < 0) return s;
  StringBuffer sb = new StringBuffer();
  int i = 0;
  while (i < ay - 1 && !Character.isLetter(s.charAt(i))) {
    sb.append(s.charAt(i)); i++;
  }
  sb.append(s.charAt(ay-1));
  sb.append(s.substring(i, ay - 1));
  int L = s.length();
  if (L - ay - 2 > 0) {
    sb.append(s.substring(ay + 2, L));
  }
  return sb.toString();
}
```

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

> He said, unconvincingly, "That's the story of my life, my dear Gambol Hedge-Bette".

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

Task 14 Timetable Clashes

Most people ran out of steam and did not attempt this one, and quite a number of the submissions were very badly coded, with errors of style such as including multiple copies of the same code "cut and pasted" into place, or attempting to reread the entire file multiple times. Here is my suggested solution, which you are encouraged to study, if only to see how to use the I/O routines and the set handling routines that were the real motivation for setting this problem in the first place in preparation for future practicals:

```
// Lecture clash checking program
// P.D. Terry, Rhodes University, 2012

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

class Subjects {
  public String name;
  public IntSet periods = new IntSet();
} // Subjects
```

```
class Clash {

  static Subjects[] offered = new Subjects[2000];

  static int findSubject(int s, int n) {
  // s is a prompt number (1 or 2), n is the number of subjects on the timetable
  // Prompts for a subject mnemonic and then returns the index of that subject in
  // the timetable record list
    int i;
    do {
      IO.write("\nSubject " + s + " (or STOP) ? ");
      String name = InFile.StdIn.readLine().trim().toUpperCase();  // clean up
      if (name.equals("STOP")) System.exit(0);
      offered[0].name = name;  // sentinel
      i = n;                   // simple linear search from top end
      while (!name.equals(offered[i].name)) i--;
    } while (i == 0);          // force them to supply a valid mnemonic
    return i;
  } // findSubject

  static void listPeriods(IntSet periods) {
  // Lists periods that are members of set periods
    for (int hex = 17; hex <= 90; hex++) {
      if (periods.contains(hex)) {
        switch (hex / 16) {
          case 1: IO.write(" Mon"); break;
          case 2: IO.write(" Tue"); break;
          case 3: IO.write(" Wed"); break;
          case 4: IO.write(" Thu"); break;
          case 5: IO.write(" Fri"); break;
          case 6: IO.write(" Sat"); break;
          case 7: IO.write(" Sun"); break;
          default: IO.write(" Bad data"); System.exit(1); break;
        }
        IO.write(hex % 16);
      }
    }
    IO.writeLine();
  } // listPeriods

  static void reportSubject(int i) {
  // Reports name and periods for subject i
    IO.write("\n" + offered[i].name + "    ");
    listPeriods(offered[i].periods);
  } // reportSubject

  static int readTimeTable() {
  // Reads timetable records and returns the number of subjects found
    final int
      mnemonic = 7,
      filler = 69;
    InFile data = new InFile("timetabl");
    if (data.openError()) {
      IO.write("timetable file could not be found");
      System.exit(1);
    }
    data.readLn();                                // ignore date line
    int n = 0;                                    // no subjects yet
    offered[0] = new Subjects();                  // prepare for later sentinel
    do {
      char ch = data.readChar();
      if (ch != ';') {                            // check for comment lines
        n++;                                      // one more subject
        offered[n] = new Subjects();
        ch = data.readChar();                     // skip column 2
        offered[n].name = data.readString(mnemonic).toUpperCase().trim();
        String skip = data.readString(filler);    // skip unwanted part
        String s = data.readWord();               // first period data
        while (!s.equals(".")) {                  // lines end with a period
          if (Character.isDigit(s.charAt(0))) {   // ignore alternative periods
            int hex = Integer.parseInt(s, 16);    // base 16 is useful here!
            offered[n].periods.incl(hex);         // add to set of fixed periods
          }
          s = data.readWord();                    // next period data
        }
      }
      data.readLn();                              // ready for next subject
    } while (!data.eof());
    return n;                                     // number of subjects found
  } // readTimeTable
```

```
  public static void main(String[] args) {
    int first, second;
    int n = readTimeTable();
    while (true) {                              // loop will terminate when STOP is read
      first = findSubject(1, n);
      second = findSubject(2, n);
      if (first != second) {                    // silly case!
        reportSubject(first);
        reportSubject(second);
        IntSet common = offered[first].periods.intersection(offered[second].periods);
        IO.write("\n" + common.members() + " clashes");
        if (common.members() != 0) listPeriods(common);
        IO.writeLine();
      }
    }
  } // main

} // Clash
```

The problem is very easily solved using sets. The slightly unusual use of a hexadecimal interpretation of the period data allows for subjects in period 10 to be handled in the same way as at any other time.

Several people did not notice that the comment lines beginning with a ; could appear anywhere within the data file, and not merely at the beginning. Several others wrote a lot of exception handling code for dealing with I/O - but the whole point of developing an I/O library is to keep all that stuff hidden in the closet and not obtruding into one's programs.

Note the simple linear search used in `findSubject`, which makes use of a so-called sentinel in the zeroth position of the array to keep the search loop simple. You will have been told that linear searches are inefficient, and so they are, but for a simple application like this they are quite adequate, and we shall use them again in the weeks ahead.

A point that was missed by some people is that the strings should be trimmed and turned into some consistent case before making the comparisons.

Finally, one group, rather than develop the program for themselves, searched for and/or found a C# decompiler and simply applied this to the EXE file that formed part of the kit. While this displayed ingenuity, and has given me a suggestion for future versions of this practical for which I am grateful, I really do wish you would try to work out solutions to practical tasks for yourselves as they have all been designed to interlock with one another. Similarly, please resist the temptation to farm out sections of the practicals so as to cut down on your apparent effort!