Computer Science 301 - 2008

Programming Language Translation

Practical for Week 19, beginning 25 August 2008 - 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 produces "Windows" compatible programs. Compilers like this are clearly designed with an "everyone has 1GB of memory and 200GB 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	Fibo Code Size	Empty Code Size	
Free Pascal	31 744	31 232	28 672	Sieve limit 16410
Optimized Free Pascal	31 744	31 232	28 672	Sieve limit 16410
Modula-2	18 609	18 098	11 946	Sieve limit 32770
Optimized M-2	18 549	18 049	11 946	Sieve limit 32770
Borland C	66 560	66 048	52 224	
Borland C++	149 504	148 480	47 104	
Watcom C	37 376	36 864	27 648	
Watcom C++	51 712	50 688	22 528	
C#	28 672	28 672	16 896	
Parva	N/A	N/A	N/A	Sieve limit 49000
Modula-2 via Borland C	62 976	62 464	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, 2008
  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");
}
```

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 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 9 - How fast/slow are various implementations?

Some times (seconds) taken to execute the various programs are shown below (figures in brackets are for a 1.04GHz laptop, others on 3.00 GHz lab machines a year or two ago). In all cases the systems ran Windows XP.

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.

(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.

Sieve: Iterations 10 000	Size of sieve 16 000	
Fibo: Upper limit 40		
	Sieve	Fibo
Free Pascal	(9.2) 4.4	(12.7) 4.2
Free Pascal (optimized)	(8.6) 4.2	(12.3) 4.1
Modula-2	(5.7) 2.4	
Modula-2 (optimized)	(2.3) 1.8	(23.5) 32.3
C	(3.8) 2.2	(12.3) 3.9
C++	(4.1) 2.1	(12.2) 4.0
Modula-2 (via C)	(25.2) 6.3	(18.9) 3.9
C#	(3.7) 3.2	(9.8) 5.1
Java (with JIT)	(6.6) 3.0	(9.3) 2.7
Java -Xint (interpret)	(74.3) 46.6	(92.6) 48.2
Parva	(1080) 448.0	(692) 277.0
Parva (optimized)	(810.0) 360.0	225.0

Task 11 Something more creative - Change your driving habits

This can be handled quite easily. The solution below illustrates the use of `substring()` as well as of `stringTokenizer()`. Points missed were that a single conversion algorithm works for conversion in either direction, and that there might be any number of dot-separated components of the name and address. Once again the program should be able to handle input data that, strictly speaking, is incorrect.

Several peopl forgot to close the output file.

```
// Convert Internet  <---> Janet addresses
// P.D. Terry, Rhodes University, 2008

import java.util.*;
import Library.*;

class Internet {

  static String convert(String address) {
  // Returns opposite form of address
    int i = address.indexOf('@');
    String user = address.substring(0, i+1);
    address = address.substring(i+1);
    StringTokenizer tokens = new StringTokenizer(address, ".", true);
    address = "";
    while (tokens.hasMoreTokens())
      address = tokens.nextToken() + address;
    return user + address;
  }
```

```
    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 address = data.readWord();
        if (data.noMoreData()) break;
        results.writeLine(convert(address));
      }

    // close results file safely - several people forgot this
      results.close();
    }

  }
```

Okay, I'll 'fess up. When I first set this exercise I thought it might be fun to use the `StringBuffer` class and the `Stack` class, and came up with this:

```
    static String convert(String address) {
    // Returns opposite form of address
      Stack st = new Stack();
      StringBuffer sb = new StringBuffer();
      StringTokenizer tokens = new StringTokenizer(address, ".@", true);
      String t;
      do {
        t = tokens.nextToken(); sb.append(t);
      } while (tokens.hasMoreTokens() && !t.equals("@"));
      while (tokens.hasMoreTokens()) st.push(tokens.nextToken());
      while (!st.empty()) sb.append(st.pop());
      return sb.toString();
    }
```

but, as already mentioned, there is always a simpler way!

Task 12 - Best of British Luck to you too, mate

Some very complicated solutions were received. It is possible to solve this problem in several ways. I think the simplest is simply 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 quite efficiently follows:

```
// Simple test bed program for checking on the generation of lucky numbers
// P.D. Terry, Rhodes University, 2008

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 not copy it 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;
}
```

```
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;
  }
}
```

Typically solutions were submitted that did rather more copying than was needed. Here is a solution based on some of those received that suffered from this.

```
// Another simple program for generating a list of lucky numbers
// P.D. Terry, Rhodes University, 2008
// Based on an idea used by several students, which does more
// copying than necessary

int remaining;

int[] produceList(int n) {
  // generate an initial sieve
  int[] temp = new int[n];
  int i = 1;
  while (i <= n) {
    temp[i-1] = i;
    i = i + 1;
  }

  // perform passes over the sieve
  int step = 2;
  remaining = n;
  while (step <= remaining) {
    int j = step - 1;
    while (j < remaining) {
      int k = j;
      while (k < remaining - 1) {
        temp[k] = temp[k + 1];
        k = k + 1;
      }
      remaining = remaining - 1;
      j = j + step - 1;
    }
    step = step + 1;
  }
  return temp;
}

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

Some students think of using a Boolean sieve idea. Here is a solution based on this which was quite neat:

```
void main() {
// Program for finding "lucky" numbers
// By Alistair Carver and Ryan Jury, 2008

  const Max = 1000;
  bool[] Lucky = new bool[Max];          // the list of numbers

  int n, N, cycle, count;   // counters

  read("Test to what number? ", N);
  if (N > Max) {
    write("N too large, sorry");
    return;
  }
```

```
  n = 0;
  while (n < Max) {
    Lucky[n] = true;
    n = n + 1;
  }

  cycle = 2;  // the number in the sequence we are removing

  // Try all cycles less than the max number
  while (cycle < N) {
    n = 0;      // current array index
    count = 0;  // counts how many numbers in the sequence we have hit

    // Only count though while the index is smaller than the max number
    while (n < N) {
      // If a number is still 'lucky', we count it
      if (Lucky[n]) count = count + 1;
      // If we have reached the number of the cycle of numbers we
      //  are removing... the number is unlucky.
      if (count >= cycle) {
        count = 0;
        Lucky[n] = false;
      }
      n = n + 1;
    }
    cycle = cycle + 1;
  }

  write("Lucky numbers between 1 and " , N, "\n");
  write("-----------------------------------\n");

  write(1);   // 1 will always be a lucky number
  n = 1;
  // Write out our lucky numbers!
  while (n < N) {
    if (Lucky[n]) write(", ", n+1);
    n = n + 1;
  }
  // DISCLAIMER: Numbers not guaranteed to actually be lucky.
  // Don't buy lottery tickets using them.
}
```

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, 2005
// Based on an idea by P. Heideman, D. Terry and M. Whitfield, 2003

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

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;
  }
}
```

Task 13 - Help out poor old Barns - Happy Snapshots

Submissions here varied from those that had not really even got started, through some very nice clean ones, to some that were almost unbelievably complicated and whose authors had sadly not sought after elegance and simplicity. There is always a simpler and more elegant solution. Believe me!

Here is my suggested solution. Besides being simpler than most submitted, it also does some error checking. Few of you thought to do this, producing solutions that might have worked if the users had treated them gently, but failing to react sensibly to silly requests to arrange 0 students, or not providing *N* names/heights after asking to arrange *N* students. I was intrigued that several people had thought of using a square root as the basis for choosing an "intelligent" layout. Perhaps the tutors had tipped them off, but if not, well done!

```
// Determine optimum layout for class photograph
// P.D. Terry, Rhodes University, 2008

import java.util.*;
import Library.*;

class Student {

  // we need their names and heights, and to record whether selected
  public String name = "";
  public double height;
  public boolean selected = false;

  public Student(double h, String s) {
    this.height = h;
    this.name = s;
  }

} // Student

class Photo {

  static Student[] students;    // static fields for convenience
  static int size;

  static Student nextTallest() {
  // Returns the next tallest available student in the list and then sets the
  // selected flag to prevent this student from being chosen again
    int maxPos = 0;
    double maxHeight = 0.0;
    for (int i = 0; i < size; i++)
      if (!students[i].selected && students[i].height > maxHeight) {
        maxPos = i;
        maxHeight = students[maxPos].height;
      }
    students[maxPos].selected = true;
    return students[maxPos];
  }

  static void writeRow(OutFile results, int n) {
  // Sets up and then displays the next row of n students
  // a typical sequence for "next" would be    4  5  3  6  2  7  1  8  0
    Student[] row = new Student[n];
    int sign = 1, step = 1, next = (n + 1) / 2 - 1; // start in the middle
    for (int i = 1; i <= n; i++) {
      row[next] = nextTallest();                    // fill this position
```

```
      next = next + sign * step;                    // and prepare for the next position
      sign = - sign;                                // swap to other side of centre
      step++;                                       // and move out one position
    }
    for (int i = 0; i < n; i++) results.write(row[i].name, 12);
    results.writeLine();
    for (int i = 0; i < n; i++) results.write(row[i].height, 12);
    results.writeLine();
    results.writeLine();
  }

  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

  // attempt to read the data
    int maxSize;
    do {
      IO.writeLine("Supply class size (>= 1)");
      maxSize = IO.readInt();                       // there may not really be that many
    } while (maxSize <= 0);                         // handle stupid abuse
    students = new Student[maxSize];                // but prepare for that number
    size = 0;
    do {
      students[size] = new Student(data.readDouble(), data.readLine());
      if (data.noMoreData()) break;                 // data ran out!
      size++;
    } while (size < maxSize);
    if (size != maxSize)                            // report the miscalculation
      IO.writeLine("Class appears to have only " + size + " students");
    if (size == 0) {                                // another safety check
      IO.writeLine("Cannot arrange a class with no students!");
      System.exit(1);
    }

  // compute the number to be put into most rows
    int rows = (int) (0.5 * (Math.sqrt(size) + 1.0));
    int onRow = size / rows;                        // most rows will have this number

  // and get on with the arrangement proper
    int toBeSeated = size;                          // still to be placed
    while (toBeSeated >= 2 * onRow) {               // for all but the front row
      writeRow(results, onRow);                     // print out that row
      toBeSeated -= onRow;                          // and prepare for the next one
    }
    writeRow(results, toBeSeated);                  // front row may be a bit longer
    results.close();                                // close the file safely
    IO.writeLine("View the map created in " + args[1]);
  }

} // Photo
```

There are a few further observations that can be made:

(a) There is no real need to sort the data before distributing it, although clearly this is a posible alternative approach. The point is that a sort into strictly ascending or descending order still only gets you part way to solving the zig-zag distribution problem.

(b) The algorithm above for distributing the data is $O(n^2)$. For small classes this would be quite acceptable. One could improve the efficiency a bit by not merely marking a student as selected, but by creating a dynamic structure and then actually eliminating a student once he or she had been selected. In this way the list of students still to be placed would get shorter and shorter on each pass. But I suspect the operation of elimination might add as much overhead as it was trying to save, unless the class were really quite big. You might like to think of how to implement this by using an `ArrayList` rather than a fixed array.

(c) I was disappointed to see that so few people thought of defining a little "class" to describe a student. And several submissions did far more string manipulation than is really necessary.

Task 14 Timetable Clashes

Some 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 re-read 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, 2008

import java.util.*;
import Library.*;

class Subjects {
  public String name;
  public SymSet periods = new SymSet();
}

class Clash {

  static Subjects[] subjects = 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 = IO.readLine().trim().toUpperCase();  // clean up
      if (name.equals("STOP") || IO.noMoreData()) System.exit(0);
      subjects[0].name = name; // sentinel
      i = n;                   // simple linear search from top end
      while (!name.equals(subjects[i].name)) i--;
    } while (i == 0);          // force them to supply a valid mnemonic
    return i;
  }

  static void listPeriods(SymSet 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();
  }

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

  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
    subjects[0] = new Subjects();               // prepare for later sentinel
```

```
    do {
      char ch = data.readChar();
      if (ch != ';') {                          // check for comment lines
        n++;                                    // one more subject
        subjects[n] = new Subjects();
        ch = data.readChar();                   // skip column 2
        subjects[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!
            subjects[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
  }

  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);
        SymSet common = subjects[first].periods.intersection(subjects[second].periods);
        IO.write("\n" + common.members() + " clashes");
        if (common.members() != 0) listPeriods(common);
        IO.writeLine();
      }
    }
  }

}
```

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 tokenizing 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. The Library routines have been carefully chosen to allow a large variety of problems to be handled without the need for the user to indulge in lots of messy error prone tokenizing and parsing, so look at them again, please.

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.