Computer Science 301 - 2014

Programming Language Translation

Practical 1, Week beginning 24 March 2014

This prac is due for submission by lunch time on your next practical day, correctly packaged in a transparent folder as usual (**unpackaged and late practical submissions will not be accepted - you have been warned**). Pracs should please be deposited in the hand-in box outside the lab. Only **one set of listings** is needed for each group, but please enclose as many copies of the cover sheet as are needed, one for each member of the group. These will be returned to you in due course.

Objectives:

In this practical you are to

- acquaint yourselves with some command line utilities, with various editors, interpreters and compilers;
- investigate various qualities of some computer languages and their implementations, including C, C++, C#, Java, Pascal and Parva.
- obtain some proficiency in the use of the various library routines that will be used later in the course.

The exercises for this week are not really difficult, although they may take longer than they deserve simply because you may be unfamiliar with the systems.

Copies of this handout, the cover sheet, the Parva language report, and descriptions of the library routines for input, output, string handling and set handling in Java and C# are available on the course web site at http://www.cs.ru.ac.za/courses/CSc301/Translators/trans.htm .

Outcomes:

When you have completed this practical you should understand

- how and where some languages are similar or dissimilar;
- how to use various command line compilers and decompilers for these languages;
- what is meant by the term "high level compiler" and how to use one;
- how to measure the relative performance of language implementations;
- the elements and limitations of programming in Parva;
- how to use I/O and set handling routines in Java.

To hand in:

This week your group is required to hand in, besides the individual cover sheets for each member:

- One copy of the listings of your solutions to the programming exercises in tasks 6, 10, 13, 14 and 15 produced by using the `LPRINT` utility from the command line (which prints listings economically).
- Electronic copies of your source code for those exercises, using the electronic submission system.
- Your commentary and solutions to the questions posed below. Part of this consists of results that you should be able to collect and record on the back of the cover sheet by the end of the first afternoon.

Keep the cover sheet and your solutions until the end of the semester. Check carefully that your mark has been entered into the Departmental Records.

You are referred to the rules for practical submission which are clearly stated in our Departmental Handbook. However, for this course pracs must be posted in the "hand-in" box outside the laboratory before the next practical session and not given to demonstrators during the session.

A rule not stated there, but which should be obvious, is that you are not allowed to hand in another student's or group's work as your own. Attempts to do this will result in (at best) a mark of zero and (at worst) severe disciplinary action and the loss of your DP. You are allowed - even encouraged - to work and study with other students, but if you do this you are asked to acknowledge that you have done so on **all** cover sheets and with suitable comments typed into **all** listings. You are expected to be familiar with the University Policy on Plagiarism, which you can consult by following a link from

http://www.scifac.ru.ac.za/ or http://www.ru.ac.za/

Before you begin

In this practical course you will be using a lot of simple utilities, and usually work at the "command line" level rather than in a GUI environment. Note in particular:

- After logging on, get to the DOS command line level by using the `Start -> Command prompt` sequence if you don't already have a shortcut (it is probably worth creating a short cut).

- Listings are conveniently produced by using the `LPRINT` command from a command window, for example

  ```
  LPRINT  Queens.cs  Queens.java
  ```

  The listings come out in a small font which enables long lines to be read easily and with narrow line spacing (so that you get more listing for your money). **Please use this utility, which prints listings in a small courier font to produce all listings submitted on this course, as it makes my job of reading the submissions much easier.** Program listings in "proportional font" are awkward to read.

- Before you can use `LPRINT` you may need to "capture" the printer, after opening a command window, by using the command `UNMAP` (if necessary) followed by `PRINTEAST` or `PRINTWEST` as appropriate.

Copies of software for home use

For this prac it is recommended that you simply work in the Hamilton lab, rather than begging, borrowing or stealing copies of a whole host of software for home use. In future pracs you will mostly use Java or C# only, and the prac kits will, hopefully, contain all the extras you need.

Task 1 (a trivial one)

We shall make use of zipped prac kits throughout the course; you will typically find sources for each week's prac in a file `pracNN.zip` on the server. Copy `prac1.zip` as needed for this week, either directly from the server on `I:\CSC301\TRANS` (or by using the WWW link on the course page), and extract the sources when you need them, into your own directory/folder, perhaps by using `UNZIP` from a command line prompt.

```
j:> copy i:\csc301\trans\prac1.zip
j:> unzip  prac1.zip
```

In the past there has occasionally been a problem with running applications generated by the C# compiler if these are stored on the network drives. If you have difficulties in this regard, for those parts of the practical that involve the use of C#, work from the local D: drive instead. After opening a command window, log onto the D: drive, create a working directory and unpack a copy of the prac kit there:

```
j:> d:
d:> md d:\G121T1111
d:> cd d:\G12T1111
d:> unzip I:\csc301\trans\prac1.zip
```

In the prac kit you will find various versions of a famous program for finding a list of prime numbers using the method known as the Sieve of Eratosthenes. You will also find various versions of a program for solving the N

Queens problem, some "empty" programs, some other bits and pieces, including a few batch files to make some of the following tasks easier and a long list of prime numbers (`primes.txt`) for checking your own!

Task 2 The Sieve of Eratosthenes in Pascal

You may not be a Pascal expert, but in the kit you will find some Pascal programs, including `SIEVE.PAS` that determines prime numbers using a Boolean array to form a "sieve". Study and compile these programs - you can do this from the command line quite easily by issuing commands like

```
FPC SIEVE.PAS
FPC FIBO.PAS
FPC EMPTY.PAS
```

to use the 32-bit Windows version of the Free Pascal compiler. Make a note of the size of the executable (use the commands `DIR SIEVE.EXE` and `DIR FIBO.EXE` and `DIR EMPTY.EXE`).

You may be able to produce a slightly faster version of the executable program for the Sieve example by suppressing the index range checks that Pascal compilers normally include for code that accesses arrays:

```
FPO SIEVE.PAS
```

How do the sizes of the executables compare? Why do you suppose the "empty" program produces the amount of code that it does?

```
PROGRAM Sieve (Input, Output);
(* Sieve of Eratosthenes for finding primes 2 <= N <= Max
   P.D. Terry, Rhodes University, 2014 *)

  CONST
    Max = 4000                                (* largest number tested *);
  TYPE
    SIEVES = ARRAY [2 .. Max] OF BOOLEAN;
  VAR
    I, N, K, Primes, It, Iterations : INTEGER  (* counters *);
    Uncrossed : SIEVES                        (* the sieve *);
  BEGIN
    Write('How many iterations '); Read(Input, Iterations);
    Write('Supply largest number to be tested '); Read(Input, N);
    IF N > Max THEN BEGIN
      WriteLn(Output, 'N too large, sorry'); HALT
    END;
    FOR It := 1 To Iterations DO BEGIN
      WriteLn(Output, 'Prime numbers between 2 and ', N);
      WriteLn(Output, '-----------------------------------');
      Primes := 0 (* no primes yet found *);
      FOR I := 2 TO N DO                      (* clear sieve *)
        Uncrossed[I] := TRUE;
      FOR I := 2 TO N DO                      (* the passes over the sieve *)
        IF Uncrossed[I] THEN BEGIN
          IF Primes MOD 8 = 0 THEN WriteLn;   (* ensure line not too long *)
          Primes := Primes + 1;
          Write(Output, I:5);
          K := I;                             (* now cross out multiples of I *)
          REPEAT
            Uncrossed[K] := FALSE; K := K + I
          UNTIL K > N
        END;
      WriteLn
    END;
    Write(Primes, ' primes');
  END.
```

Here is something more demanding:

Prime numbers are those with no factors other than themselves and 1. But the program does not seem to be looking for factors?

Look at the Pascal code carefully. How does the algorithm work (you may or may not have seen it before)? Why is it deemed to be particularly efficient? How much mental arithmetic does the "computer" have to master to

be able to solve the problem?

By experimenting with the `CONST` declaration, find out how large a sieve the program can handle. What is the significance of this limit? *Hint*: you should find that funny things happen when the sieve gets too large, though it may not immediately be apparent. Think hard about this one!

Task 3 The Sieve in C or C++

The kit also includes C and C++ versions of these programs. Compile these and experiment with them in the same way with the 32-bit Windows compliers:

```
BCC SIEVE.C          (using the Borland compiler in C mode)
BCC SIEVE.CPP        (using the Borland compiler in C++ mode)
CL  SIEVE.C          (using the WatCom compiler in C mode)
CL  SIEVE.CPP        (using the WatCom compiler in C++ mode)
```

Once again, make a note of the size of the executables, and in particular, compare them with the earlier versions. Can you think of any reason why the differences are as you find them?

Task 4 Jolly Java, what

As should be familiar, you can compile a Java program directly from the command line with a command like

```
javac Sieve.java       (using the JDK compiler)
```

Task 5 See C#

You can compile the C# versions of these programs from the command line, for example:

```
csharp Sieve.cs
```

(You may have to do this on the local D: drive) Make a note of the size of the ".NET assemblies" produced (`SIEVE.EXE, EMPTY.EXE` and `FIBO.EXE`). How do these compare with the other executables?

Task 6 Progress to Parva

On the course web page you will find a description of Parva, a toy language very similar to C, and a language for variations on which we shall develop a compiler and interpreter later in the course. The main difference between Parva and C/Java/C# is that Parva is stripped down to bare essentials.

Learn the Parva system by studying the language description where necessary, and trying the system out on the supplied code (`SIEVE.PAV, FIBO.PAV` and `QUEENS1.PAV`). There are various ways to compile Parva programs. The easiest is to use a command line command:

```
parva Sieve.pav          simple error messages
parva -o Sieve.pav       slightly optimized code
parva -l Queens.pav      error messages merged into listing.txt
```

You may have to do this on the local D: drive.

Hand in listings of your final corrected programs produced with the `LPRINT` command.

Task 7 A blast from the past - some 1980s vintage 16 bit DOS compilers

We have some early compilers, two of which you might like to explore.

These compilers will not run directly on 64 bit Windows systems with 4 GB of memory. They were constructed at a time when 64 KB of memory was considered "large", and as such they are masterpieces of software engineering.

We can run 16 bit software in various ways. The simplest - adequate for our purpose - is to run the DOS 6.2 emulator known as `DosBox` as a Windows application. To do this, first copy a shortcut to your desktop. The shortcut can be found by navigating to the `I:\utils` folder. Once you have it on the desktop, clicking it will open an 80 x 25 text window, set up a few paths to executables, and present you with a DOS prompt.

(Advance warning - sometimes the mouse pointer disappears when you are using DOSBox. The Mouse does not work within the system at all. If you lose the mouse, CTRL+F10 usually gets it back again).

At this prompt you can execute various familiar DOS commands, like `DIR` and `DEL`, but you cannot execute 32 bit software designed for Windows. No matter - you can edit files on the D: drive using 32 bit software like NotePad++, and they will be visible in the DosBox window for further processing.

Turbo Pascal 6.0

Start by recompiling the Pascal source code mentioned previously, executing the code, and making the same measurements as before, using commands like

```
TPC SIEVE.PAS
TPC FIBO.PAS
TPC EMPTY.PAS
TPC QUEENS.PAS
```

and comment on any major differences that you notice from your use of Free Pascal. TPC executes a version of Turbo Pascal last released in 1990, by which stage the Pascal language it compiled was quite a lot more complex than the original language of 1970.

Turbo Pascal 1.0

For some real fun, try out the early Turbo Pascal system, by giving the command

```
TURBO
```

This system is all contained in 39 KB, and that includes the compiler, a full screen editor, and runtime support. Once the first screen loads you can import a source file, then press C to compile it and R to run the compiled program.

Everything - even the object code - is kept in RAM, which partly explains the blazing speed. To save the machine code version as a .COM file (another blast from the past) you will have to use the `Options` available in a fairly obvious way.

Task 8 High level translators

It may help amplify the material we are discussing in lectures if you put some simple Parva programs through a high-level translator, and then look at, and compile, the generated code to see the sort of thing that happens when one performs automatic translation of a program from one high-level language to another.

We have a home-brewed system that translates Parva programs into Java. The system is called `Parva2ToJava`. It is still under development - meaning that it has some flaws that we might get you to repair in a future practical. Much of the software for this course has been designed expressly so that you can have fun improving it!

You can translate a Parva program into Java using a command of the form exemplified by

```
Parva2ToJava Sieve.pav
Parva2ToJava Fibo.pav
Parva2ToJava Empty.pav
```

A Java source file is produced with an obvious name; this can then be compiled with the Java compiler by using commands of the form:

```
javac Sieve.java
javac Fibo.java
javac Empty.java
```

and executed with commands of the form

```
java Sieve
java Fibo
java Empty
```

Take note of, and comment on, such things as the kind of Java code that is generated (is it readable; is it anything like you might have written yourself?), and of the relative ease or difficulty of using such a system. You might also like to comment on the performance of the programs that are produced, by running timing tests as before.

Another program in the kit is a variation on the example found in the book on page 90. This has an intentional weakness. See if you can spot it in time before May 7, and help get us all safely into the next 20 years of our democracy.

Run the Parva compiler directly:

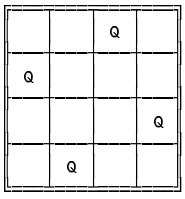
```
Parva voter.pav
```

Then try translating the program to Java and compiling and running that:

```
Parva2ToJava voter.pav
javac voter.java
java voter
```

Task 9 The N Queens problem

In the kit you will find various equivalent programs that attempt to solve the famous *N Queens* problem. These use a back-tracking approach to determine how to place N Queens on an N * N chess board in such a way that no Queen threatens or is threatened by any other Queen - noting that a Queen threatens another Queen if the two pieces lie on a common vertical, horizontal or diagonal line drawn on the board. Here is a solution showing how 4 Queens can be placed safely on a 4 * 4 board:



Compile one or more of these programs and try them out. For example

```
FPC  QUEENS.PAS
QUEENS
```

There are two versions written in each of Pascal, Java and C#. One version uses parameters to pass information between the routines, the other version uses global variables. At some stage you could usefully spend a little time studying `Tutorial 1` on the web site, which explains the technique behind the solution.

In the kit you will also find some variations on Parva programs for solving the N-Queens problem. Not all of these programs are correct. Try compiling them with the Parva compiler first, and observe the outcome. Then try converting them to Java (without editing them in any other way), observe the outcome, and compile and run the Java programs, such as are produced. Do the Parva programs need to be acceptable to the Parva compiler if they are to be acceptable to `Parva2ToJava`? What can you learn from this exercise about using a tool of this nature? Have we made `Parva2ToJava` "as simple as possible, but no simpler"? Do we have to, or could we, make it simpler still? Do we have to make it more complex? Why - or why not?

Summarize your thoughts in a short essay which should form part of your submission.

Task 10 - How fast/slow are various language implementations?

Different compilers - even for very similar programs - can produce code of very different quality. In particular "interpretive" systems (of which the Parva implementation is one example) produce programs that run far more slowly than do "machine" or "native" code systems. Carry out some tests to see these effects for yourselves, and how severe they are, by comparing the execution times of some of the programs.

Summarize your findings on page 2 of the cover sheet, explaining briefly how you come to the figures that you quote. Is Java better/worse than C# (the source code in each case is almost identical)? Do 16-bit compilers fare better or worse than 32-bit compilers?

Hint: the machines in the Hamilton Labs are *very* fast, so you should try something like this: modify the programs to comment out nearly all the output statements (since you are not interested in seeing the solutions to the Fibonnaci sequence a zillion times, or a zillion lists of prime numbers, or measuring the speed of I/O operations), and then run the programs and time them with a stop watch. Choose sizes for the sieve or chessboard (and a suitable number of iterations) that will produce measurable times of the order of a few seconds (don't try to time very fast programs this way - deliberately use the repeat counts to increase the execution time).

Although Java is often touted as being an interpreted language, in fact the latest versions of the Java "interpreter" - the program executed when you give the `java` command - actually indulge in "just in time" compiling (see textbook page 32) and "JIT" the code to native machine code as and when it is convenient - which results in spectacularly improved performance. It is possible to frustrate this by issuing the `java` command with a directive `-Xint`:

```
javac Sieve.java
java -Xint Sieve
```

to run the program in interpretive mode. Try this out as part of your experiment.

Task 11 - Reverse Engineering and Decompiling

In lectures you were told of the existence of decompilers - programs that can take low-level code and attempt to reconstruct higher level code. There are a few of these available for experiment.

`jad`	a decompiler that tries to construct Java source from Java class files
`javap`	a decompiler that creates pseudo assembler source from a Java class file
`gnoloo`	a decompiler that creates JVM assembler source from a class file
`oolong`	an assembler that creates Java class files from JVM assembler source
`ildasm`	a decompiler that creates CIL assembler source from a .NET assembly
`ilasm`	an assembler that creates a .NET assembly from CIL assembler source
`peverify`	a tool for verifying .NET assemblies

Try out the following experiments or others like them:

(a) After compiling `Sieve.java` to create `Sieve.class`, decompile this:

```
jad   Sieve.class
```

and examine the output, which will appear in `Sieve.jad`

(b) Disassemble `Sieve.class`

```
javap  -c Sieve >Sieve.jvm
```

and examine the output, which will appear in `Sieve.jvm`

(c) Disassemble `Sieve.class`

```
gnoloo Sieve.class
```

and examine the output, which will appear in `Sieve.j`

(d) Reassemble `Sieve.j`

```
oolong Sieve.j
```

and try to execute the resulting class file

```
java Sieve
```

(e) Be malicious! Corrupt `Sieve.j` - simply delete a few lines in the section that corresponds to the *Main* function (use lines with opcodes on them). Try to reassemble the file (as above) and to re-run it. What happens?

(f) Compile `Sieve.cs` and then disassemble it

```
csharp Sieve.cs
Disassemble Sieve            (calls ildasm from a batch file, produces Sieve.cil)
```

and examine the output, which will appear in `Sieve.cil`

(g) Reassemble `Sieve.cil`

```
Reassemble Sieve             (calls ilasm from a batch file, produces new Sieve.exe)
```

and try to execute the resulting class file

```
Sieve
```

(h) Be malicious! Corrupt `Sieve.cil` - simply delete a few lines in the section that corresponds to the *Main* function (use lines with opcodes on them). Try to reassemble the file (as above) and to re-run it. What happens?

(i) Experiment with the .NET verifier after step (g) and again after step (h)

```
NetVerify Sieve              (calls peverify from a batch file)
```

Task 12 - A Cautionary Tale

A few years ago I set some simple programming exercises for this class to do in this practical, and provided an executable version of a solution to assist the class in understanding the problem. I had not reckoned with the guile of some of your predecessors who, rather than write their own program, decompiled my executable and handed that in instead. Caution: After years of experience I can spot fraudulent behaviour very quickly. I'd thought I was safe because I had not let the class know about .NET decompilers, but my colleague Professor Google had obviously been consulted. The group pointed me very quickly to a tool written by JetBrains, known as dotPeek. This is very easy to use and quite fun, so I have installed it on the lab machines temporarily for use in this practical.

As a variation on task 11, try using it to decompile `Sieve.exe`, `Fibo.exe` and `Queens.exe`:

```
dotPeek Sieve.exe              (runs dotPeek from a batch file)
dotPeek Fibo.exe               (runs dotPeek from a batch file)
```

and navigate to the decompiled source code. Go on to save this under different names (for example `Sieve2.cs` and `Queens2.cs`) and then recompile these sources to see if you can get working executables.

What happens if you try to decompile an executable that was not produced from a .NET compatible compiler? Try it.

Caution: Don't try to run tools like this to decompile programs I might give you in executable form! Nevertheless, I am sure that you can see that they might have a great deal of use - for example in legitimately recreating source that might have got lost. I have not, myself, explored the options of dotPeek farther than I needed to make the suggestions above, but feel free to experiment.

Time to think for yourselves

And now for something completely different! (where have you heard that before?)

- and don't use a search engine!

Nothing you have done so far should have extended your programming talents very much. To get the old brain cells working a little harder, turn your minds to the following.

It is important that you learn to use the IO libraries `InFile, OutFile` and `IO`. These will be used repeatedly in this course. Please do not use other methods for doing I/O, or spend time writing lots of exception handling code.

Pat Terry's problems are sometimes reputed to be hard. They only get very hard if you don't think very carefully about what you are trying to do, and they get much easier if you think hard and spend time discussing the solutions with the tutors or even the Tyrant himself. His experience of watching the current generation of students suggests that some of you get beguiled by glitzy environments and think that programs just "happen" if you can guess what to click on next. Don't just go in and hack. It really does not save you any time, it just wastes it. Each of the refinements can be solved elegantly in a small number of lines of code if you think them through carefully before you start to use the editor, and I shall be looking for elegant solutions.

Remember a crucial theme of this course - "Keep it as simple as you can, but no simpler".

I am looking for imaginative, clear, simple solutions to the problems.

Task 13 One way of speeding up tedious recursion

An easy one to start you off.

Consider the following program, which is the Parva version of one of those in the kit: (`fIBO.PAV`):

```
// Print a table of Fibonacci numbers using (slow) recursive definition
// P.D. Terry,  Rhodes University, 2014

  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;
    return fib(m-1) + fib(m-2);
  } // fib

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

If you compile and run this for a fairly large *Limit* you will easily see that it takes longer and longer as *i* increases. In fact, the algorithm is easily shown to be $O(1.6^N)$.

Of course, if all you want is a simple table of Fibonacci numbers and not a text-book demonstration of a recursive function, it is much simpler and faster to proceed as follows (`fib1.pav`):

```
// Print a table of Fibonacci numbers using (fast) iterative method
// P.D. Terry,  Rhodes University, 2014

  void main() {
    int
      term   = 0,
      first  = 0,
      second = 1,
      limit;

    read("Supply upper limit ", limit);
    write(term, "\t", first, "\n");
    while (term < limit) {
      term = term + 1;
      write(term, "\t", second, "\n");
      int next = first + second;
      first  = second;
      second = next;
    } // while
  } // main
```

or, perhaps, with the *while* loop expressed using only two variables, as follows (`fib2.pav`):

```
    while (term < limit) {
      term = term + 1;
      write(term, "\t", second, "\n");
      second = first + second;
      first  = second - first;
    }  // while
```

Well, suppose we *do want* a recursive function (perhaps *you* don't, but this is not a democracy!). The reason that the program runs ever slower is, of course, that evaluating the function for a large argument effectively sets up a tree of calls to functions whose values have already been computed previously many, many times. Try drawing the tree, if you need convincing.

A way of improving on this is as follows. Each time a value is computed for a hitherto unused value of the argument, store the result in a (global) array indexed by the value of the argument as well as "returning" it. Then, if another call is made for this value of the argument, obtain the value of the function from the array rather than making the two interior recursive calls. So, for example, if one had evaluated `fib(2)` … `fib(5)` the array would contain:

0	1	1	2	3	5	0	0	0	0	0	0	0	.....
0	1	2	3	4	5	6	7	8	9	10	11	12	.....

and evaluating `fib(6)` would amount to a single addition only, while evaluating `fib(10)` at this point still be much faster than in the simple version in `Fibo.pav` (why?).

Wait! This is a very well known exercise. You are bound to be able to find many solutions on the web. Resist the temptation. Work out your own solution, please.

(The same goes for some other exercises in this course. You will get far more out of them if you try them by yourselves, or within your group, rather than hunting for previous solutions.)

This sort of optimization can be very useful in cases where a recursive function might be called many thousands of times in a system for values of an argument within a known range (so that the array size is easily fixed). The technique has a special name. After you have developed your solution, see if you can find out what this name is.

Task 14 Creative programming - How long will my journey take?

You will need to become acquainted with various library classes to solve the next two tasks, which are to be done in Java or C# as you prefer, and are designed to emphasize some useful techniques.

Descriptions of the relevant I/O library routines can be found on the course website, and a simple sample program using some of the library routines can be found in the kit as the program `SampleIO.java` (listed below) or `SampleIO.cs` (essentially the same). Note the precautionary error checking.

It is important that you learn to use the IO libraries `InFile, OutFile` and `IO`. These will be used repeatedly in this course. Please do not use other methods for doing I/O, or spend time writing lots of exception handling code or using other ways to extract tokens from complete lines.

A local bus service (like those in big cities such as Grahamstown) links a large number of stops. Suppose we are given the travel time from any one stop to the next (assume that this time would be the same for return journeys) and that this information is captured in a long list of times given in minutes, with the stop names (abbreviated if necessary to 8 letters) between them. For example, if we had 8 stops we might have a list like

```
College  8  Hamilton  5  Oakdene  12  Gino's  25  Mews 9  Union  12  Steers 17  Athies
```

Viv's Bus Service want to put up a poster at each stop from which one can easily determine the total travel time between any one stop and any other one. For the data given this might look like this (see file `BUSES`):

	College	Hamilton	Oakdene	Gino's	Mews	Union	Steers	Athies
College	0	8	13	25	50	59	71	88
Hamilton	8	0	5	17	42	51	63	80
Oakdene	13	5	0	12	37	46	58	75
Gino's	25	17	12	0	25	34	46	63
Mews	50	42	37	25	0	9	21	38
Union	59	51	46	34	9	0	12	29
Steers	71	63	58	46	21	12	0	17
Athies	88	80	75	63	38	29	17	0

Write a program to create such a table. Use the `ArrayList` class to store the original data - you will need a small auxiliary class to record the successive pairs of names and travel times - and then set up a two-dimensional matrix to contain the computed values (note that this matrix will be *symmetric*).

Task 15 Creative programming - Goldbach's conjecture

Goldbach's conjecture is that every even number greater than 2 can be expressed as the sum of two prime numbers. Write a program that examines every even integer *N* from 4 to *Limit*, attempting to find a pair of prime numbers (*A* , *B*) such that $N = A + B$. If successful the program should write *N, A* and *B*; otherwise it should write a message indicating that the conjecture has been disproved. This might be done in various ways. Since the hidden agenda is to familiarize you with the use of a class for manipulating "sets", you **must use a variation on the sieve method** suggested by the code you have already seen: create a "set" of prime numbers first in an object of the `IntSet` class, and then use this set intelligently to check the conjecture.

Demonstration program showing use of InFile, OutFile and IntSet classes

This code is to be found in the file `SampleIO.java` in the prac kit. There is an equivalent C# version in the file `SampleIO.cs`.

```
import library.*;

class SampleIO {

  public static void main(String[] args) {
  //                                      check that arguments have been supplied
      if (args.length != 2) {
        IO.writeLine("missing args");
        System.exit(1);
      }
  //                                      attempt to open data file
      InFile data = new InFile(args[0]);
      if (data.openError()) {
        IO.writeLine("cannot open " + args[0]);
        System.exit(1);
      }
  //                                      attempt to open results file
      OutFile results = new OutFile(args[1]);
      if (results.openError()) {
        IO.writeLine("cannot open " + args[1]);
        System.exit(1);
      }

  //                                      various initializations
      int total = 0;
      IntSet mySet = new IntSet();
      IntSet smallSet = new IntSet(1, 2, 3, 4, 5);
      String smallSetStr = smallSet.toString();

  //                                      read and process data file
      int item = data.readInt();
      while (!data.noMoreData()) {
        total = total + item;
        if (item > 0) mySet.incl(item);
        item = data.readInt();
      }
  //                                      write various results to output file
      results.write("total = ");
      results.writeLine(total, 5);
      results.writeLine("unique positive numbers " + mySet.toString());
      results.writeLine("union with " + smallSetStr
                        + " = " + mySet.union(smallSet).toString());
      results.writeLine("intersection with " + smallSetStr
                        + " = " + mySet.intersection(smallSet).toString());
    } // main

  } // SampleIO
```