Computer Science 301 - 2015

Programming Language Translation

Practical 1, Week beginning 24 August 2015 - Solutions

The submissions received were very varied in quality. There was some innovative work handed in, but there was evidence that people had missed several important points. You can find complete C# source versions of the program solutions in the solution kit `PRAC1A.ZIP` on the server.

Some general comments:

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

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

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

(d) Some submissions were received that had not made proper use of the `IO`, `InFile` and `OutFile` classes *as you had been told to do*. These library classes are designed to make text based I/O as simple as possible without having to deal with buffered readers, scanners, exceptions, string tokenizers and all that stuff that you were probably subjected to in CSC 102, but without realizing that the best thing to do with bizarre code is to hide its details in a well-designed library. Have a look at the solutions below, where hopefully you will see that the I/O aspects have been made very simple indeed.

(e) Please remember to use the `LPRINT` facility for producing source listings economically. In later practicals the listings will get very wide, and they are hard to read if they wrap round.

Tasks 2 to 6 - The Sieve of Eratosthenes

The Pascal compilers use 16-bit `INTEGER` arithmetic (values from -32768 .. 32767), although they allow very large array sizes, as arrays can also be indexed in some compilers by so-called `LONGINT` (32 bit) variables. And, in fact (probably comes as a surprise to you C-language types), Pascal also allows arrays to have negative indices, so that one can declare, for example

```
VAR PopulationOfRome     : ARRAY [-45 .. 320] OF INTEGER; (* an array with 366 elements *);
    BigArrayOfRealValues : ARRAY [0 .. 65534] OF REAL;    (* an array with 65535 elements *)
```

However, an array indexed by an INTEGER variable cannot access an element whose subscript is greater than 32767.

Although the Sieve size in the supplied code was apparently "large enough" the Sieve algorithm as supplied could and did easily collapse when applied to a search for large primes, using variables of the standard INTEGER type. Consider the code:

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

When `I` becomes large enough, `K+I` will eventually become larger than 32767, but if the overflow is not trapped it means that the sum appears to go negative (think back to your CSC 201 course). If you set N to be 20000 this happens for the first time after detecting the prime number 16411, so that the maximum effective sieve algorithm with the code above seems to be limited to primes from 2 to 16411. If you set N to be 32000 it happens for the first time after detecting the prime number 863: in due course the multiple 38*863 is "crossed off" and `K` tries to advance to 32794.

We can extend the range of the algorithm by a trick which I did not really expect you to discover, but which is worth pointing out. Simply replace the above code by something which at first looks ridiculous:

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

To get this to work you have to set a compiler directive to switch range checking off (using a command line parameter, or inserting a pragma/directive something like {$R-} at the start of the source code). Some students might have got intrigued by all this and probed further (well done). If you try interesting things like "turn off the range checks" on the program as originally supplied, the algorithms appear to allow you to generate higher prime numbers. Trouble is, they might not do it properly, and you find that for some "bigger" values of `Max` you actually seem to find fewer prime numbers generated.

Learning to program in "non-bondage" languages like C++ is like trying to learn to drive in a car without brakes - very exciting, you go faster and faster, and then you die, sooner or later. Fortunately C# and Java are much safer.

Come on - there must be a better way (there always is). How can the algorithm be changed so that range checks can be left enabled and the system find large primes without bombing?

The C, C++, C# and Parva compilers use 32 bit integers, and thus don't seem to have this problem (or at least, it is much harder to reproduce), but, of course, the amount of real memory available to them may be limited. And how many people noted that the C and C++ source code had declared the size of the array incorrectly?

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.

Executable sizes and limits (as obtained on my machine running Win-7, 32 bit version)

	Empty	Fibo	Sieve	Queens Parameters	Queens1 Globals	Maximum prime < 32 000 (without trick)	Number of primes < 20 000
Free Pascal FPC (.EXE files)	27716	30276	31300	31812	31812	16411	breaks
Turbo Pascal TP6 (.EXE files)	1472	2640	3248	3808	3632	16411	breaks
Turbo Pascal TP60 (.EXE files)	1472	2640	3136	3472	3328	16411	2199 x
Turbo Pascal 1 (.COM files)	11386	11601	11971	12285	12154	16411	2199 x
C# (.EXE files)	33280	33792	33792	34304	34304	31991	2262
Parva	N/A	N/A	N/A	N/A	N/A	31991	2262
BCC .C files (.EXE files)	52224	66048	66560	N/A	N/A	31991	2262
BCC .CPP files (.EXE files)	47104	148480	149504	N/A	N/A	31991	2262
CL .C files (.EXE files)	21504	34816	34816	N/A	N/A	breaks	2262
CL .CPP files (.EXE files)	21504	50176	50688	N/A	N/A	breaks	2262

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, although this runs slowly when the checks are present.

The Borland 5.5 and WatCom C/C++ compilers are designed for 32 bit integers and 32-bit operating systems, 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. 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.

Task 6 - The Sieve in Parva

The corrected code is very simple. A few groups got it badly wrong, with braces in the wrong places. Note that the body of a *do-while* loop has to be executed at least once, which means that the code should really have been transformed to achieve this. However, even if this is not done it "works". Why?

```
// Sieve of Eratosthenes for finding primes 2 <= n <= Max (Parva version)
// P.D. Terry,  Rhodes University, 2015

void Main() {
  const Max = 32000;
  bool[] uncrossed = new bool[Max];          // the sieve
  int i, n, k, it, iterations, primes = 0;   // counters
  read("How many iterations? ", iterations);
  bool display = iterations == 1;
  read("Supply largest number to be tested ", n);
  if (n > Max) {
    write("n too large, sorry");
    return;
  }
  write("Prime numbers between 2 and " , n, "\n");
  write("-----------------------------------\n");
  it = 1;
  while (it <= iterations) {
    primes = 0;
    i = 2;
    while (i <= n) {                         // clear sieve
      uncrossed[i] = true;
      i = i + 1;
    }
    i = 2;
    while (i <= n) {                         // the passes over the sieve
      if (uncrossed[i]) {
        if (display && (primes - (primes/8)*8 == 0))
          write("\n");                       // ensure line not too long
        primes = primes + 1;
        if (display) write(i, "\t");
        k = i;                               // now cross out multiples of i
        uncrossed[k] = false;
        k = k + i;
        while (k <= n) {
          uncrossed[k] = false;
          k = k + i;
        }
      }
      i = i + 1;
    }
    it = it + 1;
    if (display) write("\n");
  }
  write(primes, " primes");
} // Main
```

Task 7 - High level translators

Some of what might be perceived as "unreadability" presumably relates to the fact that `Parva2ToCSharp` is obliged to translate the read and write multiple parameter functions into a collection of equivalent IO operations from the supporting library.

It is easy to trip up the process. Consider the silly program and its apparently correct translation:

```
void Main () {
  int b;
  bool a = b > 4;
} // Main
```

```
using Library;

class Wrong {

  static public void Main(string[] args) {
    int b;
    bool a = b > 4;
  } // Main

} // Wrong
```

The C# compiler will detect that the assignment statement is meaningless, as `b` has not been initialized, but a Parva compiler and the `Parva2ToCSharp` converter are not as sophisticated.

You may not have seen the point that using a tool like this would allow you to develop and maintain your programs in Parva and then simply convert them to C# when you want to get them compiled on some other system (perhaps so that they can run quickly). So normally a user of `Parva2ToCSharp` would not read or edit the C# code at all. Because of this it is not necessary for the converted code to incorporate the original comments.

If you had played with the `Voter.pav` examples properly you should have found that if all the ages supplied are below 18 there is an attempted division by zero reported. If the program is translated into C#, the same data will generate a corresponding exception. If one were using `Parva2ToCSharp` as a way of speeding up execution one would perhaps be confused by an error message that did not relate back to the original source. And the C# compiler can warn of a variable that is declared but never used, which the Parva one cannot.

Task 8 - Number of solutions to the N Queens problem (Queens2)

I imagine that everyone managed to get this to run, and could correct the few mistakes in the Parva versions.

1	2	3	4	5	6	7	8	9	10	11
1	0	0	2	10	4	40	92	352	724	2680

12	13	14	15	16
14200	73712	365596	2279184	14772512

Task 9 - How fast/slow are various implementations?

Some times (seconds) taken to execute the various programs are shown below, as measured on my computer, which runs Win7-32 (and so could get proper timings for the 16 bit systems as well). Times measured in the lab might have been rather different.

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. Some of 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. There was a script `timer.bat` in the kit which some people tried using to overcome these problems, but it won't work under DosBox

(d) There are clearly some anomalies here. There is something very odd about programs compiled with range checks using Turbo Pascal 6.0, and I have no idea what it can be. However, the general effects are apparent - modern compilers make better use of the large opcode sets on modern processors and produce faster programs suited to the operating systems on those machines. The experiments with the Queens programs show that using global variables is marginally faster then passing parameters (nobody commented on this as I recall, which was a pity). Pascal programs running without range checks execute faster than

the same programs compiled to include range checks.

(e) The times taken by the 16-bit systems running under the DOSBox emulator in the lab probably showed quite clearly the adverse effects of emulation. This system was apparently developed to allow game freaks to run "old" computer games designed in the 8086, 80386 and 80486 era to run on modern computers running at vastly greater clock speeds. There must be better ways of running old number-crunchers on the latest operating systems, and this will be investigated further in the future if I get the chance.

Timings

	Sieve Number of iterations 10000 Maximum number tested 15000	Queens Number of iterations 10000 (parameter used) Board size 8	Queens1 Number of iterations 10000t (global variables used) Board size 8
Free Pascal FPC (.EXE files)	2.85 ($R+) 2.45 ($R-)	1.64 ($R+) 1.20 ($R-)	1.62 ($R+) 1.09 ($R-)
Turbo Pascal TP6 (.EXE files)	27.25 ($R+) something odd!	20.30 ($R+)	17.02 ($R+)
Turbo Pascal TP60 (.EXE files)	1.75 ($R-)	3.86 ($R-)	1.96 ($R-)
Turbo Pascal 1 (.COM files)	3.09 ($R+) 2.67 ($R-)	6.67 ($R+) 6.54 ($R-)	2.99 ($R+) 2.55 ($R-)
C# (.EXE files)	0.83	1.05	1.03
Parva (standard)	25 for 1000 (250 for 10000)	11.8 for 1000 (118 for 10000)	11.6 for 1000 (116 for 10000)
Parva (optimized)	17.8 for 1000	8.4 for 1000	8.5 for 1000
Parva2ToCSharp (.EXE files)	0.78	1.08	1.03
BCC .C files (.EXE files)	0.67	N/A	N/A
BCC .CPP files (.EXE files)	0.69	N/A	N/A
CL .C files (.EXE files)	(I don't have this compiler)	N/A	N/A
CL .CPP files (.EXE files)		N/A	N/A

Of course, it is fairly dangerous to draw conclusive results from such a crude set of tests and limited sample, but the main effects show up quite clearly.

Task 12 Creative Parva programming - the "hailstone" sequence

Suppose N is a positive number that starts a sequence defined by the following rules: If a term M is odd, the next term in the sequence is $3M + 1$. If a term M is even, the next term is $M / 2$. The sequence terminates when $M = 1$. For example, the sequence that starts with $N = 6$ is as follows:

6 3 10 5 16 8 4 2 1

and in this particular case the length of the sequence is 9. You were asked to write a function procedure that, given N, will return the length L of the sequence beginning with N, and then to continue to write a little program that used this function to determine the smallest positive integer N that produces a sequence length L greater than K, where K is a number used as input data. For example, for $K = 12$, the result should be $N = 7$. 7 is the smallest positive integer that generates a sequence with more than 12 members.

A hailstone sequence is easily generated, save for the annoyance of not having the `%` (modulo) operator and an *else* for the *if* statement. Never mind - in a few weeks time we shall modify Parva to have those extravagances…

One simple solution follows

```
/* Solve the hailstone series problem
   P.D. Terry, Rhodes University, 2015 */

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;
} // LengthOfSeries

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");
} // Main
```

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 the following):

```
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);
} // LengthOfSeries
```

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 13 Something more creative - "Pig Latin" in C#

A naïve 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 specified as a program "arg: and storing the result in another text files specified by another "arg). There was no need to tokenize, simply to use the `ReadWord` method in my library).

I do not recall many submissions that used the file opening methods correctly, unfortunately

```
// Convert an English text to "Pig Latin"
// Words may only contain letters
// P.D. Terry, Rhodes University, 2015

using Library;
using System;
using System.Text;

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[0] + "ay";
    return s;
  } // Convert

  public static void Main(string[] args) {
  // first check that command line arguments have been supplied
  // attempt to open data file    using file names from "args" ++++++++++++++++++++++++++
  // attempt to open results file and check that the files open correctly +++++++++++++++
  // all this as in SampleIO.cs - 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();
  } // Main

} // ToLatin1
```

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[ay-1] + s.Substring(0, ay - 1);
    return s;
  } // Deconvert
```

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:

> She 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 `StringBuilder` 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!!"
  StringBuilder sb = new StringBuilder();
  int i = 0, sl = s.Length;
  while (i < sl && !Char.IsLetter(s[i])) {
    sb.Append(s[i]); i++;
  }
  if (i < sl) {
    char first = s[i];
    i++;
    while (i < sl && Char.IsLetter(s[i])) {
      sb.Append(s[i]); i++;
    }
    sb.Append(first);
    sb.Append("ay");
    while (i < sl) {
      sb.Append(s[i]); i++;
    }
  }
  return sb.ToString();
} // Convert

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 < 1) return s;
  StringBuilder sb = new StringBuilder();
  int i = 0;
  while (i < ay - 1 && !Char.IsLetter(s[i])) {
    sb.Append(s[i]); i++;
  }
  sb.Append(s[ay-1]);
  sb.Append(s.Substring(i, ay - i - 1));
  int L = s.Length;
  if (L - ay - 2 > 0) sb.Append(s.Substring(ay + 2, L - ay - 2));
  return sb.ToString();
} // Deconvert
```

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

> He replied, sadly, "That's very clever of you to find so many, my dear Gambol Hedge-Bette".

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

Task 14 Something more creative - Another look at the infamous Sieve in C#

Here is one possible solution to the problem posed - making use of two sets, both initially empty. One will contain the numbers that are found to be prime, and the other will contain the numbers that cannot be prime - the multiples of the prime numbers.

Note that we have used the `ToString()` method to display the set of primes. This is for demonstration purposes, really. The simple `ToString()` method in my library does not display the set elements split nicely into lines. You might like to see if you can find a way to improve on this.

```
// Sieve of Eratosthenes making use of two sets rather than an array of Booleans
// P.D. Terry,  Rhodes University, 2015

using Library;

class SieveSet {

  public static void Main(string [] args) {
    int limit = IO.ReadInt("Supply largest number to be tested ");
    IntSet primeSet = Primes(limit);
    IO.WriteLine(primeSet);
    IO.WriteLine(primeSet.Members() + " primes");
  } // Main
```

```
  static IntSet Primes(int max)  {
  // Returns the set of prime numbers smaller than max
    IntSet primeSet = new IntSet();   // the prime numbers
    IntSet crossed  = new IntSet();   // the sieve
    for (int i = 2; i <= max; i++) {  // the passes over the sieve
      if (!crossed.Contains(i)) {
        primeSet.Incl(i);
        int k = i;                    // now cross out multiples of i
        do {
          crossed.Incl(k);
          k += i;
        } while (k <= max && k > 0);
      }
    }
    return primeSet;
  } // Primes

} // SieveSet
```

If you merely want to count the primes you need but one set:

```
// Sieve of Eratosthenes making use of one set rather than an array of Booleans
// Merely count the primes
// P.D. Terry,  Rhodes University, 2015

using Library;

class PrimeCount {

  public static void Main(string [] args) {
    int limit = IO.ReadInt("Supply largest number to be tested ");
    IO.WriteLine(NumberOfPrimes(limit) + " primes");
  } // Main

  static int NumberOfPrimes(int max)  {
  // Returns the number of prime numbers smaller than max
    int count = 0;
    IntSet crossed  = new IntSet();   // the sieve
    for (int i = 2; i <= max; i++) {  // the passes over the sieve
      if (!crossed.Contains(i)) {
        count++;
        int k = i;                    // now cross out multiples of i
        do {
          crossed.Incl(k);
          k += i;
        } while (k <= max && k > 0);
      }
    }
    return count;
  } // NumberOfPrimes

} // PrimeCount
```

There are some other variations on these programs in the solution kit.

Task 15 Something more creative - A simple graphics class in C#

The code below shows possible modifications to the `Queens2` program that was supplied. This stores the coordinates of the various points needed for the graph, before opening a graphics window and drawing a sequence of line segments and circles to depict the graph.

The scaling and shifting used here in the lines reading

```
        if (solutions == 0) d = 0.0; else d = Math.Log10(solutions);  // fudge
        IO.WriteFixed(d, 0, 2);
        IO.WriteLine(solutions);
        xg[n] = 40 * n;                                 // fairly arb scaling!
        yg[n] = 600 - (int) (Math.Round(80 * d));       // invert!
```

was a bit of a hack to get the graph to fill the window more or less completely. You will hopefully have seen that the solution count does rise pretty exponentially after the bit of noise at the bottom (that is, for very small boards), which was simply fudged away to avoid problems with evaluating the log of zero. Another way of getting around this problem is simply to start plotting from $n = 4$.

```
     using Library;
     using System;
**   using GraphicsLib;

     class Queens2 {

     // ..... as before

       public static void Main(string[] args) {
         const int Max = 16;
**       double d;
         bool[] a = new bool[Max + 1];
         bool[] b = new bool[2 * Max + 1];
         bool[] c = new bool[2 * Max + 1];
         int[] x  = new int[Max + 1];
**       int[] xg = new int[Max + 1];
**       int[] yg = new int[Max + 1];
**       for (int n = 1; n <= Max; n++) {
           solutions = 0;
           for (int i = 1; i <= n; i++)      a[i] = true;
           for (int i = 1; i <= 2 * n; i++)  b[i] = c[i] = true;

           Place(1, n, a, b, c, x);

           IO.Write("Board size "); IO.Write(n, 2);
           IO.Write(" Solutions ");
**         if (solutions == 0) d = 0.0; else d = Math.Log10(solutions);  // fudge
**         IO.WriteFixed(d, 0, 2);
**         IO.WriteLine(solutions);
**         xg[n] = 40 * n;                                  // fairly arb scaling!
**         yg[n] = 600 - (int) (Math.Round(80 * d));        // invert!
         } // for
**
**       // The prompt will allow you to move the command window clear of the graphics window
**
**       IO.WriteLine("Hit Enter to Start");
**       IO.ReadLine();
**       GraphicsWindow gw = new GraphicsWindow(600, 600);
**
**       // Draw the graph as a set of line segments and mark each point with a small circle
**
**       for (int i = 1; i <= Max - 1; i++) {
**         gw.DrawLine(xg[i], yg[i], xg[i+1], yg[i+1]);
**         gw.DrawCircle(xg[i], yg[i], 10);
**       }
**       gw.DrawCircle(xg[Max], yg[Max], 10);
**
**       // The prompt and response allow you to gaze at the graph before killing the program
**
**       IO.WriteLine("Hit Enter to Exit");
**       IO.ReadLine();
       } // Main

     } // Queens2
```