E-GENTING PROGRAMMING COMPETITION 2007

PUBLIC LECTURE

DECEMBER 2007

LECTURE NOTES

Second draft
Jonathan Searcy
6 December 2007

Table of Contents

1 INTRODUCTION

1.1 Format of the Lecture

Ladies and Gentlemen, welcome ...

Today we will try and discover how to win the next E-Genting Programming Competition by looking at how the questions in last year's paper could have been solved.

Last year's paper contained four questions. The contestants could answer one or more questions. The questions had varying levels of difficulty.

The questions were:

Question	Description	Marks
1.	**Financial Statements** A commercial reporting program with configurable parameters.	240
2.	**The Bankcard Kid** A diagnostic program that emulates the user of an automatic teller machine.	160
3.	**Merge and Sort** A trivial exercise in merging and sorting.	80
4.	**Air Pollution Monitor** A problem in interpolating between points in two dimensions and the application of performance-enhancing algorithms.	320

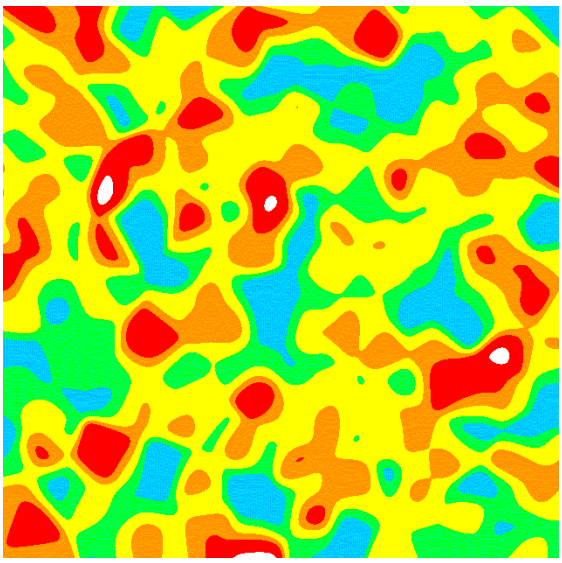
I will start this year with the most interesting problem, the Air Pollution Monitor. It applies a simple algorithm for drawing smooth curves between points that I first used for drawing pump performance curves in the late seventies.

Next we will talk about the Merge and Sort problem. This is a problem so easy that if you can't do it you should not only demand your university fees back, you should demand your high school fees back too.

We will then move on to look at the Financial Statements problem and the question most contestants attempted, the Bankcard Kid.

2 AIR POLLUTION MONITOR

2.1 Introduction

E10126033N00259537194 E10149349N00255301342 E10147289N00252330231 E10128254N00258029010 E10134282N00313547133 E10146512N00306312421 E10133274N00303149145 E10129093N00300315194 E10123597N00311052107 E10135371N00311513370 :	

The objective of the Air Pollution Monitor is to create a program that can be executed on a web browser that converts a table of statistics of the kind shown on the left side of this slide into an air pollution map similar to that on the right hand side of the slide. The air pollution map uses different colours to represent different levels of air pollution like hypsometric tints on a topographical map. The program must dynamically refresh the air pollution map every 30 seconds.

The statistics on the left hand side of the slide are a longitude and latitude to precisions of tenths of a second, and a three-digit air pollution index value produced by an air pollution sensor located at the co-ordinates. Can anyone give me an estimate of how far a tenth of a second of latitude is in metres at the Earth's surface? (Approximately 3 metres)

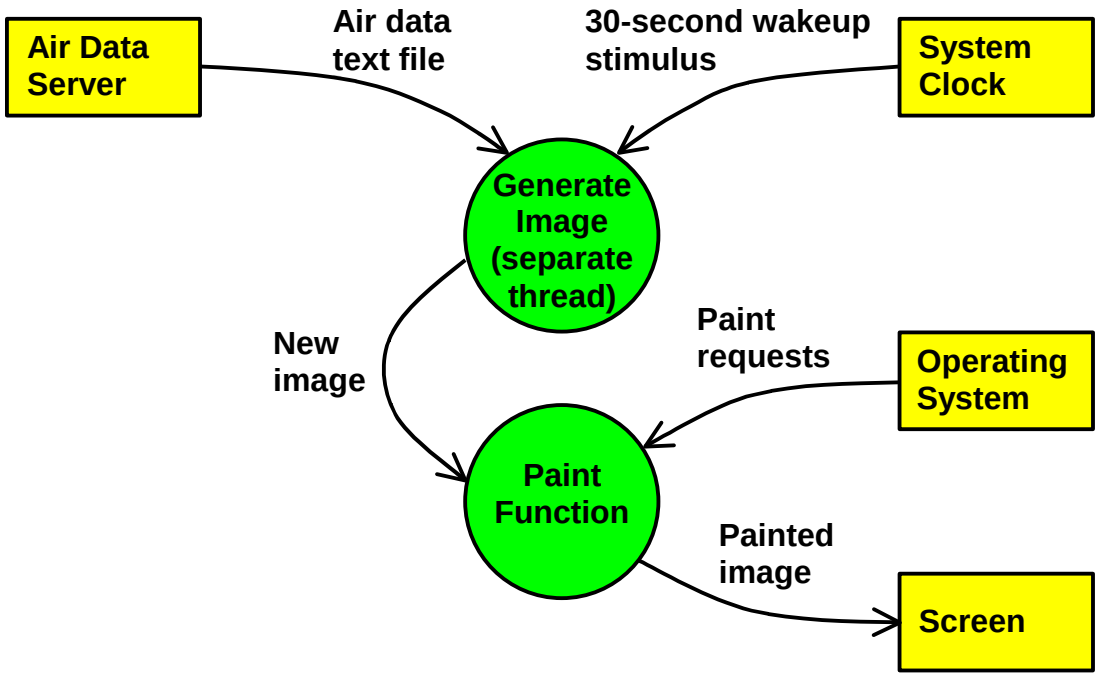
The question paper provides us with some formulas that we can use to estimate the air pollution index at any point x-y from the air pollution index values in the text file. So in theory all we have to do is to apply the formulas for each pixel to estimate the air pollution index at the pixel and then colour the pixel depending on the value of the air pollution index.

But life is never that easy in the E-Genting Programming Competition. We are told that the program should be able to convert the table into the map within 30 seconds, with a preferred conversion time of less than 5 seconds. We are also told that a naïve implementation that evaluates the formulas for every pixel takes approximately 51 seconds to generate a new image. To do well in this question we need to find a way to make the program at least twice as fast and preferably over 10 times as fast as the naïve implementation.

But nevertheless, the question paper tells us that we will get half-marks for a naïve implementation, which is the same value as the much more popular Bankcard Kid

question, for only a fraction of the amount of handwriting, so it's got to be worth trying. So let us look first of all at what is needed to accomplish a naïve implementation.

2.2 High level Data Flows



The question paper tells us that the client program should show the old air pollution map while it is generating a new air pollution map.

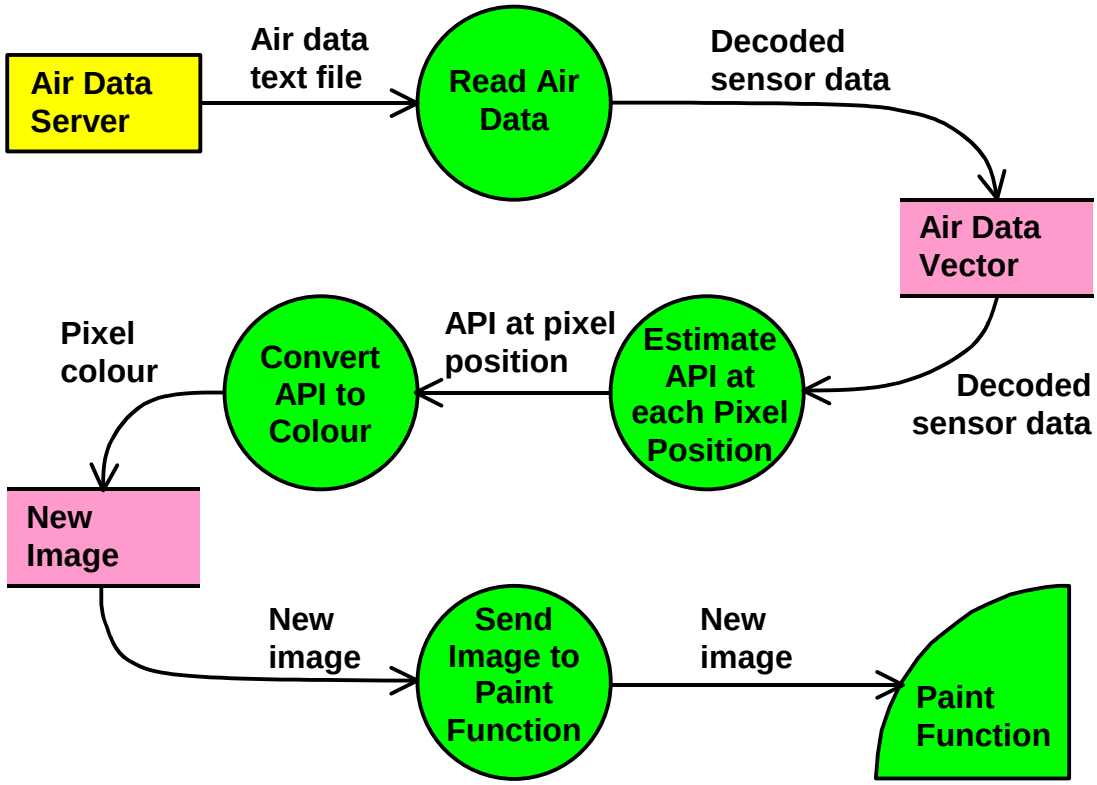
This suggests that we need a background thread generating a new image, while the paint function is still using the old image to render the screen, which may need to be done at virtually any time as a consequence of application windows being moved around the physical screen.

So we need a background thread that sleeps until the 30-second refresh time arrives and then reads the latest version of the air pollution data and generates a new image.

The Java paint function should wait to be called by the operating system. When it is called, it should draw the latest image on the screen.

In terms of concurrency, there's not much more to this problem.

2.3 Generating an Image

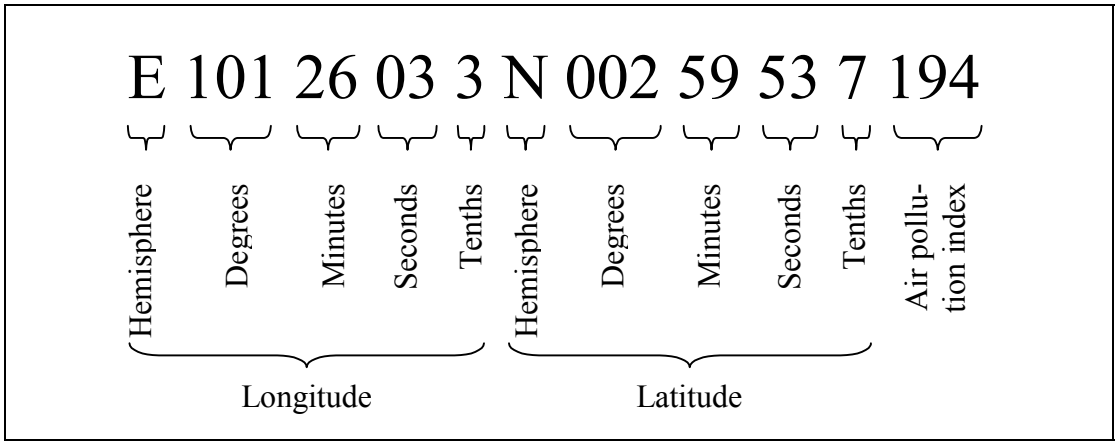


To generate each image, we start with a process that reads the latest air pollution data from the server and converts the data into a form that is convenient for the interpolation function to use; say a vector or array of x-y co-ordinates and air pollution index values.

We then need a function that estimates the API for each pixel on the screen by applying the formulas given in the question paper. This function can then pass the values of the air pollution index to another function that converts the API values into their corresponding colour codes and writes the pixel colour to the new image.

When all the pixels in the image have been coloured in we need to pass the completed image to the paint function so that the paint function can use it for rendering the image on the screen.

2.4 Text File Format



The question paper tells us that the format of each line of air pollution data file has the format shown in this slide.

The first character is a code 'E' for east or 'W' for west to indicate whether the longitude is east or west of Greenwich. We can think of this code as a sign indicator. If we want to use a right-handed Cartesian co-ordinate system, longitudes west of Greenwich are negative and longitudes east of Greenwich are positive.

Next we have the longitude of the sensor in degrees, minutes, seconds and tenths of seconds.

The latitude field also starts with a hemisphere indicator, which is either 'N' for north or 'S' for south. Again, we can think of northern latitudes as positive and southern latitudes as negative.

After the latitude indicator is the value of the latitude in degrees, minutes, seconds and tenths of seconds.

Last there is the three-digit value of the air pollution index measured by the sensor.

2.5 Reading the Text File

```
airDataUrl = new URL ("http",
    "genting.com.my", "rnd/airdata.txt");
airDataCon = (HttpURLConnection)
    airDataUrl.openConnection();
airDataStream = airDataCon.getInputStream ();
//...
// Read the air pollution data
//...
airDataStream.close ();
```

In Java, setting up an input stream to read the air pollution data from the server is quite easy.

We start by creating a URL instance that identifies the location of the air pollution data on the Internet.

We then create a connection to the server and get an input stream for the connection.

The program can then read from the input stream in the same way as it might read from System.in.

When the program has finished with the input stream it should close the stream to release the resources associated with the connection.

2.6 Reading Lines of Input

```
// Read the longitude hemisphere

ch = airDataStream.read();
if (ch == -1) return END_OF_FILE;
if (ch != 'E' && ch != 'W')
    throw new RuntimeException ();
longHemi = ch;

// Read the longitude

dms = new StringBuffer();
for (i = 0; i < 8; i++) {
    ch = airDataStream.read();
    if (ch == -1) return END_OF_FILE;
    if ( ! Character.isDigit ((char)ch))
        throw new RuntimeException ();
    dms.append ((char)ch);
}
longTenths = dmsToTenths (dms.toString());
if (longHemi == 'W') longTenths = -longTenths;
```

There are simple ways and complicated ways to read lines of text. The simple way is to make the sequence of the code follow the sequence of the data to be received. The complicated way is to contort the code to make full use of inbuilt library functions. I prefer the simple way. It might generate a few more source instructions, but it always works.

This slide is an edited excerpt from the sample solution to the Air Pollution Monitor problem posted on the web after the competition. The first half reads and validates the longitudinal hemisphere indicator and the second half reads and validates the 8 digits in the longitude value field.

You can see how the code follows the sequence of the input. First the program reads the hemisphere indicator and then it reads each of the 8 digits in the longitude value, validating them as it goes along.

If the program strikes and end-of-file character in the input stream, identified by character value –1, it passes the end-of-file condition back to its caller. If it identifies an invalid character in the input stream, it throws a runtime exception.

At the end of the module it converts the degrees, minutes and seconds field into simple tenths of seconds so that the program can do calculations on the longitude without having to convert between degrees, minutes and seconds on the fly.

Lastly, the hemisphere is incorporated into the longitude value by making longitudes west of Greenwich negative.

2.7 Converting Degrees, Minutes and Seconds to a Simple Scalar Quantity

```
private int
dmsToTenths (
    String     dms)     // Degrees, minutes and seconds
{
    int        tenths;  // Tenths of seconds
    tenths = Character.digit (dms.charAt(0), 10);
    tenths = tenths * 10 +
          Character.digit (dms.charAt(1), 10);
    tenths = tenths * 10 +
          Character.digit (dms.charAt(2), 10);
    tenths = tenths * 6 +
          Character.digit (dms.charAt(3), 10);
    tenths = tenths * 10 +
          Character.digit (dms.charAt(4), 10);
    tenths = tenths * 6 +
          Character.digit (dms.charAt(5), 10);
    tenths = tenths * 10 +
          Character.digit (dms.charAt(6), 10);
    tenths = tenths * 10 +
          Character.digit (dms.charAt(7), 10);
    return tenths;
}
```

This is how the example solution converted the degrees, minutes, seconds and tenths field into a scalar quantity of tenths of degrees.

The Character.digit function converts a character into its numeric equivalent. It converts the digit ‘0’ into the value zero, the digit ‘1’ into the value one and so forth.

The variable tenths is first loaded with the value of the high-order digit of the degrees field.

In the next statement, it is multiplied by ten to convert it into tens of degrees and added to the value of the tens of degrees digit, which is also in units of tens of degrees. Single units of degrees are processed in the same way.

When the program comes to processing tens of minutes, it converts the tenths variable, which was previously in units of degrees, into units of tens of minutes by multiplying it by six. It then adds in the tens of minutes and so on until the result comes out in units of tenths of seconds.

The whole process of converting the degrees, minutes and seconds string into scalar tenths of seconds can be thought of as a sequence of unit conversions and the adding in of digit values.

2.8 API Interpolation Formula

$$api(x,y) = \frac{\sum_{i=1}^{n} ad_i \, w(x,y,x_i,y_i)}{\sum_{i=1}^{n} w(x,y,x_i,y_i)}$$

Where:

api(x,y)	is the estimated API at point (x,y).
x	is a longitude in minutes.
y	is a latitude in minutes
n	is the number of sensors.
ad_i	is the API reported by sensor i in airdata.txt.
xi	is the longitude in minutes of sensor i.
yi	is the latitude in minutes of sensor i.
$w(x,y.x_i,y_i)$	is the weighting function defined below.

The question paper gives us this formula for estimating the API at a particular point x-y.

Note that if all the weights are all one, the estimated API is simply the average of all the APIs. The effect of the weighting function is simply to adjust the frequency of each sensor's result in the calculated average. It gives greater significance to the samples closer to the point we need the estimate for. The estimated API is merely a weighted average.

Converting the formula into a program is quite straightforward.

2.9 API Interpolation Function

```
double
estimateApi (
     AirData[]      airDataArr,    // Air data array
     double         x,             // Longitude
     double         y)             // Latitude
{
     double         wt;            // Weight of sample
     double         sumApiWt = 0;  // Sum of API*wt
     double         sumWt = 0;     // Sum of wt
     for (int i = 0; i < airDataArr.length; i++) {
          wt = weight (x, y,
               airDataArr[i].x, airDataArr[i].y);
          sumApiWt += airDataArr[i].api * wt;
          sumWt += wt;
     }
     if (sumWt == 0) return 0;
     return sumApiWt / sumWt;
}
```

This is a function that evaluates the earlier formula.

The function accepts a reference to an array of structures that contain the data in the air pollution data file and a pair of x-y co-ordinates that identify the point for which the API is to be estimated.

All the function has to do is to evaluate the weight of the sample by calling a function that implements the weighting formula and then totalling up the product of the API values and the weight and the weights.

Technically, we should do something about a potential divide-by-zero, in the event all the weights are zero. In the sample solution the program assumed that no weights meant unpolluted air. Perhaps separate colour for status unknown would have been a better alternative.

2.10 Weighting Formula

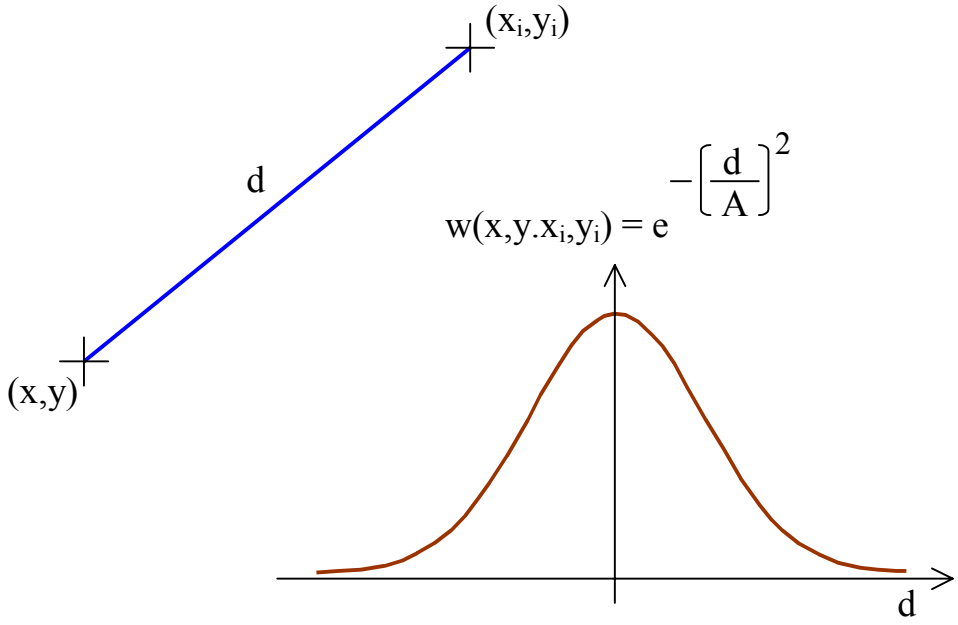
$$w(x,y.x_i,y_i) = e^{-\left(\frac{((x_i - x)^2 + (y_i - y)^2)}{A^2}\right)}$$

Where:

$w(x,y.x_i,y_i)$	is the value of the weighting function.
x	is a longitude in minutes.
y	is a latitude in minutes
x_i	is the longitude in minutes of sensor i.
y_i	is the latitude in minutes of sensor i.
e	is Euler's number (approximately 2.7128)
A	is the constant 0.9 nautical miles.

The question paper also gives us the formula for the weighting function.

2.11 Weighting Curve



If we look at a point x-y and a sensor position x_i-y_i, and let the distance between the points be 'd', then the weighting formula reduces to $e^{-(d/A)^2}$.

If we plot this curve, we will see that the weighting function is a bell-shaped curve similar to the normal distribution. The further x_i-yi is from x-y, the less significance is given to the sample at x_i-yi.

2.12 Converting the Constant A

The question paper tells us 'A' is the constant 0.9 nautical miles.

The question paper also tells us a nautical mile is a minute of longitude at the equator.

But if we are working in tenths of seconds, we need to convert the constant 'A' from nautical miles into tenths of seconds.

A = 0.9 nautical miles

= 0.9 minutes of longitude

= 0.9 x 60 seconds of longitude

= 0.9 x 60 x 10 tenths of seconds of longitude

= 540 tenths of seconds of longitude

Where we do need to be careful is in dealing with the units of the constant 'A'. 'A' is given as 0.9 nautical miles. The question paper tells us that a nautical mile is a minute of longitude at the equator. So if we are working in units of tenths of seconds, we must convert A into tenths of seconds by multiplying it by 60 to convert from minutes into seconds and then by 10 to convert from seconds into tenths of seconds. Giving us a value of A in tenths of seconds of 0.9 by 600 or 540.

2.13 Weighting Function

```
double
weight (
     double    x,   // Longitude of point
     double    y,   // Latitude of point
     double    xi,  // Longitude of sample
     double    yi)  // Latitude of sample
{
     return Math.exp (
          -((xi-x)*(xi-x) + (yi-y)*(yi-y))
               / (540*540));
}
```

If x and y are in units of tenths of seconds, the weighting function is simply this.

The Math.exp function evaluates the value of e^x.

The easiest way to calculate the square of a number is to just multiply the number by itself. Any self-respecting compiler will only evaluate the difference once and then multiply the difference by itself.

2.14 Scaling the Image

Required scale is 1:330,000.

This means one inch on the map must correspond to 330,000 inches in the Klang Valley.

Tookit.getScreenResolution gives us the screen resolution in pixels per inch, let us call it 'R'.

The inter-pixel distance on the map is 1/R.

Therefore the inter-pixel distance in the Klang Valley is 330,000/R inches.

The question paper tells us a minute of longitude at the equator is 72913 inches.

Therefore the inter-pixel distance in tenths of seconds is (330,000*60*10)/(R*72913).

If R is 96, the inter-pixel distance is approximately 28 tenths of a second.

Now we know how to estimate the API at an arbitrary point, we need to find a way to relate pixels in an image to the corresponding points in the physical Klang Valley.

The question paper tells us that the air pollution map should be rendered at a scale of approximately 1:330,000. This means that one inch on the map should correspond to 330,000 inches in the Klang Valley.

But what we really need to know is the number of tenths of seconds the distance between two pixels corresponds to.

To perform this calculation, we can use the getScreenResolution method in the Toolkit class that returns the screen resolution in pixels per inch.

The inter-pixel distance is the reciprocal of the resolution.

We can then apply the required scale to find out the corresponding distance in the Klang Valley in units of inches and then we can convert inches to tenths of seconds using the constants provided in the question paper.

2.15 Calculating the Image Dimensions

```
interPix = /* Inter-pixel distance */;
fromLong = dmsToTenths("10120000");
toLong = dmsToTenths ("10150000");
imageWidth = (toLong – fromLong) / interPix;
fromLat = dmsToTenths ("00250000");
toLat = dmsToTenths ("00320000");
imageHeight = (toLat – fromLat) / interPix;
```

Before we can create the image, we need to work out the size of the image in pixel units.

On the previous slide I explained how to calculate the inter-pixel distance. We can assume that this has been loaded into a variable called interPix.

The question paper tells us that all the sensors lie between longitudes 101° 20’ and 101° 50’ east and latitudes 002° 50’ and 003° 20’ north.

We can use the dmsToTenths function we created earlier to convert the longitude limits into scalar tenths of degrees. We can then take the difference between the two divided by the inter-pixel distance to figure out the width of the image in pixels.

We can work out the height of the image by doing the same calculations on the latitude values.

2.16 Filling In the Image

```
newImage = createImage (imageWidth, imageHeight)
g = newImage.getGraphics ();
for (int x = 0; x < imageWidth; x++) {
     for (int y = 0; y < imageHeight; y++) {
          rx = x * interPix;
          ry = y * interPix;
          api = estimateApi (airDataArr, rx, ry);
          pixColor = apiToColor (api);
          g.setColor (pixColor);
          g.fillRect (x, imageHeight-y-1, 1, 1);
     }
}
```

To fill the image, we create an image with the required dimensions and obtain a graphics context for the image.

We then make the program loop through each pixel.

The program then converts the pixel co-ordinates into their corresponding co-ordinates in the Klang Valley.

We can use the estimateApi function we designed earlier to estimate the air pollution index at that point in the Klang Valley.

Now we find we need a function to convert the API at that position in the Klang Valley into the colour that is to be shown on the screen. We will figure out the workings of this function shortly.

Having worked out the colour to be shown, we can use the setColour and fillRect functions to set the colour of the pixel in the image.

When all the pixels have been coloured, the program can pass the completed image to the paint function for painting on the user screen.

2.17 Colour Table

API	Status	RGB Colour Components		
		Red	Green	Blue
0-50	Good	0	204	255
51-100	Moderate	0	255	64
101-200	Unhealthy	255	255	0
201-300	Very unhealthy	255	153	0
301-500	Hazardous	255	0	0
Above 500	Emergency	255	255	255

The question paper gives us the colours to be shown on the map when the API is between various limits.

2.18 Converting the API into a Colour

```
Color
apiToColor (
     double    api)      // API value
{
     if (api <= 50)
          return Color (0, 204, 255);
     if (api <= 100)
          return Color (0, 255, 64);
     if (api <= 200)
          return Color (255, 255, 0);
     if (api <= 300)
          return Color (255, 153, 0);
     if (api <= 500)
          return Color (255, 0, 0);
     return Color (255, 255, 255);
}
```

As I said, it's trivial.

2.19 Passing the New Image to the Event Dispatch Thread

```
try {
    EventQueue.invokeAndWait
          (new Refresher(newImage));
}
catch (InterruptedException e) {
    throw e;
}
catch (Exception e) {
    throw new RuntimeException (e.toString());
}
```

All that remains is to pass the image from the image generation thread to the event dispatch thread so that it can be painted on the screen.

In the sample solution I did this by using the EventQueue.invokeAndWait method. The invokeAndWait method blocks until the request reaches the head of the event queue. It then processes the request in the event processing thread and returns control back to the application.

The synchronisation could also have been accomplished by using the Java synchronized statement to control access to a shared variable.

This completes the first half of the Air Pollution Monitor question: getting it to work at all. Now we must find ways to make it fast.

2.20 Memory/Speed Trade Off

> For a given function, the more space, the faster. This is true over an amazingly large range.
>
> Professor Fredrick Brooks, 1975.

In 1975, the year before I started programming, Professor Brooks observed that there is a trade-off between space and speed.

Once we have gotten rid of any stupid inefficiency, the easiest way to make a computer program faster is to use more memory to avoid redundant calculations.

Basically, if we have more memory, we can store intermediate values that might be needed in future calculations so that they don't need to be recalculated when those future calculations need to be made.

2.21 Weighting Function Lookup Table

Pixels away (x and y)	Weighting Function Values				
	-2	-1	0	1	2
-2	0.000	0.001	0.003	0.001	0.000
-1	0.001	0.055	0.234	0.055	0.001
0	0.003	0.234	1.000	0.234	0.003
1	0.001	0.055	0.234	0.055	0.001
2	0.000	0.001	0.003	0.001	0.000

The sample solution uses two optimisations to improve the speed of the program.

The first is that it uses a table to evaluate the weighting function instead of executing the calculations. Evaluating e^x is not the fastest of operations, especially when the arguments involve some arithmetic as well. If we can replace these operations with an integer arithmetic lookup table, we can expect to achieve a significant improvement in speed.

To accomplish this, the sample solution does not work in tenth-of-second spatial units. Instead it works in pixel units. The latitude and longitude of the sensors are rounded off to the nearest integer x-y pixel co-ordinates.

Then for any given pixel we can use a table similar to the one above to calculate the weighting function value. For example, if the sensor is one pixel position above the pixel the program is colouring, then the weight of the sensor will be 0.234.

It is not strictly necessary to store the whole table as it is shown here. The sample solution stores only a quadrant of the table and uses the absolute value of the difference between the x-y co-ordinates of the sensor and the x-y co-ordinates of the pixel to look up the weighting function value.

Even that is a little wasteful. All that is really needed is a triangular sector of the table, but that involves a slightly more complicated array indexing function.

The table does not need to be the size of the whole map. The question paper tells us that sensors with weighting function values less than 10^{-6} may be disregarded. This means that we can limit the size of the table to distances with weighting function values greater than or equal to 10^{-6}.

2.22 Calculating the Range of the Weighting Function

```
weightRange = 0; // In pixel units
do {
    weightRange ++ ;
    w = weight (weightRange*interPix, 0);
} while (w >= 1.0e-6);
```

The easiest way to calculate the range of the weighting function is to simply search progressively increasing values until the value of the weighting function falls below 10^{-6}.

2.23 Creating the Weight Table

```
double weightTable[];
weightTable = new double[weightRange*weightRange];
for (x = 0; x < weightRange; x++) {
    for (y = 0; y < weightRange; y++) {
        weightTable[y*weightRange + x] =
            weight (0, 0, x*interPix, y*interPix);
    }
}
```

Creating the weight table is simply a matter of creating an array and loading it using the mathematical weighting function we programmed earlier.

Because Java does not support two-dimensional arrays, but rather arrays of arrays, I have use a hard-coded array indexing function to implement the two-dimensional array.

2.24 Using the Weight Table

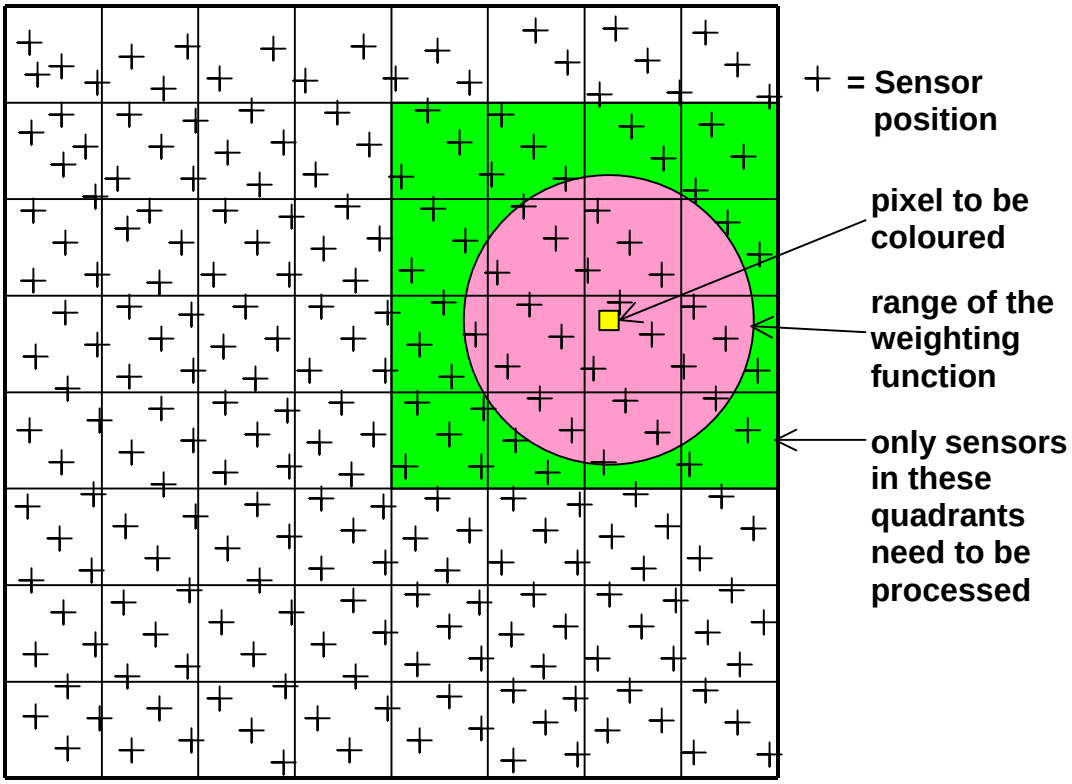
```
double
quickWeight (
    int         x,      // Horizontal position of pixel
    int         y,      // Vertical position of pixel
    int         xi,     // Horizontal position of sample
    int         yi)     // Vertical position of sample
{
    int         dx, dy;
    dx = x > xi ? x – xi : xi – x;
    dy = y > yi ? y – yi : yi – y;
    if (dx >= weightRange || dy >= weightRange) return 0;
    return weightTable [dy*weightRange + dx];
}
```

Using the weighting table is a simple matter of calculating the absolute value of the distance between the pixel and the sample. If the distance is greater than the weight range, then the returned weight can be rounded to zero.

If the sample is within the weight range, the weight value can be quickly determined by looking it up in the table.

This optimisation produced a performance improvement of a factor of 4 in the sample solution.

2.25 Restricting the Scope of the Sum



The second optimisation that was used in the sample solution was to reduce the number of samples that need to be processed for each pixel by dividing the mapped area into a grid of quadrants.

If we divide the area of the map into quadrants, it is relatively easy, and computationally fast, to determine the quadrants that lie within the range of the weighting function from the pixel. It is simply a matter of figuring out the quadrants of the pixel position plus and minus the range limits in the vertical and horizontal directions.

If we store the sensor data in separate chains for each quadrant, the program only needs to process the sensor data in the quadrants that fall within the range of the weighting function. A two-dimensional array can be used to store chain headers, one header for each quadrant. Each header can point to a linked list that contains references to the sensor data for each sensor located in the quadrant.

2.26 Creating the Quadrant Data Chains

```
// Initialise the grid headers and footers

for (gx = 0; gx < gridWidth; gx++)
    for (gy = 0; gy < gridHeight; gy++)
        gridHeaders[gy*gridWidth + gx] = -1;

// Put the sensors into the appropriate
// grid quadrants

for (i = 0; i < airDataArr.length; i++) {
    gx = airDataArr[i].x / QUAD_SIZE;
    gy = airDataArr[i].y / QUAD_SIZE;
    gi = gy * gridWidth + gx;
    airDataArr[i].next = gridHeaders[gi];
    gridHeaders[gi] = i;
}
```

This is how the grid chains were loaded in the example solution.

The grid headers were initialised to –1, which means end-of-chain.

The quadrants of the sensors were found by dividing the x and y co-ordinates of the sensor by the quadrant size.

A grid index was found using the algorithm for creating a two-dimensional array from a one-dimensional array.

And then the sensor was added to the chain for the quadrant.

The headers and ‘next’ members in the air data array form a unidirectional linked list of the sensors that are located in the quadrant.

2.27 Determining the Quadrants to be Scanned

```
gxMin = (x – weightRange) / QUAD_SIZE;
if (gxMin < 0) gxMin = 0;
gxMax = (x + weightRange) / QUAD_SIZE + 1;
if (gxMax > gridWidth) gxMax = gridWidth;
gyMin = (y – weightRange) / QUAD_SIZE;
if (gyMin < 0) gyMin = 0;
gyMax = (y + weightRange) / QUAD_SIZE + 1;
if (gyMax > gridHeight) gyMax = gridHeight
```

The next thing we need to do is to determine which quadrants need to be processed.

This is a simple matter of determining the limits of the range of the weighting function in units of grid quadrants, which is a simple matter of dividing the limit of the weight range by the quadrant size.

In the case of the upper limit, we must add one to ensure that the last quadrant is not truncated.

2.28 Using the Quadrant Data Chains

```
sumApiWt = 0;
sumWt = 0;
for (gx = gxMin; gx < gxMax; gx++) {
    for (gy = gyMin; gy < gyMax; gy++) {
        for (
            i = gridHeaders[gy*gridWidth + gx];
            i != -1;
            i = airData.next
        ) {
            airData = airDataArr[i];
            wt = weight (x, y, airData.x, airData.y);
            sumApiWt += wt * airData.api;
            sumWt += wt;
        }
    }
}
if (sumWt == 0) return 0;
return sumApiWt / sumWt;
```

This is how the quadrant data chains were used in the sample program.

The program loops through each chain header in the x and y directions.

It then loops through each sensor in the quadrant, calculates the weight associated with the sensor, and then accumulates the sums.

Finally, if there are no sensors within range of the pixel, it returns zero instead of crashing the program with a divide-by-zero error.

Implementing this optimisation in the sample solution produced a performance improvement of a factor of around 10.

2.29 Summary

At this stage I would like to highlight that the program fragments I have shown today are slightly different to the equivalent program fragments in the sample solution. This is mainly because the sample solution was designed to solve the problem, whereas the program fragments I have presented today were designed to show how to solve the problem, which is a slightly different problem.

The Air Pollution Monitor is an exercise in programming to a specification that gives the programmer an opportunity to show his creativity and problem solving abilities by devising effective optimisations. It exercises these programming techniques:

- Using threads and concurrency to display one image while another is being generated.
- Reading data from an HTTP server.
- Reading fields from a text file.
- Unit conversion.
- Programming mathematical formulas.
- Using the graphical functions of the language and operating system.
- Using memory to obtain speed.
- Using lookup tables to improve speed.
- Using a grid to reduce the volume of data that needs to be processed.

Would anyone like to ask any questions about the Air Pollution Monitor problem?

3 MERGE AND SORT

3.1 Introduction

```
======================== drs/GuiComp.cpp 37.2 ========================
14a15
> // 24-08-07   JS      Changed 'comp issue loc date' to 'shift date'
145c146
<       frm.FedText (COL_WIDTH*2, l, "Comp issue loc date");
---
>       frm.FedText (COL_WIDTH*2, l, "Shift date");
======================== drs/GuiIncd.cpp 37.2 ========================
14a15
> // 24-08-07   JS      Changed 'comp issue loc date' to 'shift date'
146c147
<       frm.FedText (COL_WIDTH*2, l, "Comp issue loc date");
---
>       frm.FedText (COL_WIDTH*2, l, "Shift date");
```

The Merge and Sort program had to process change listings of the type shown on this slide.

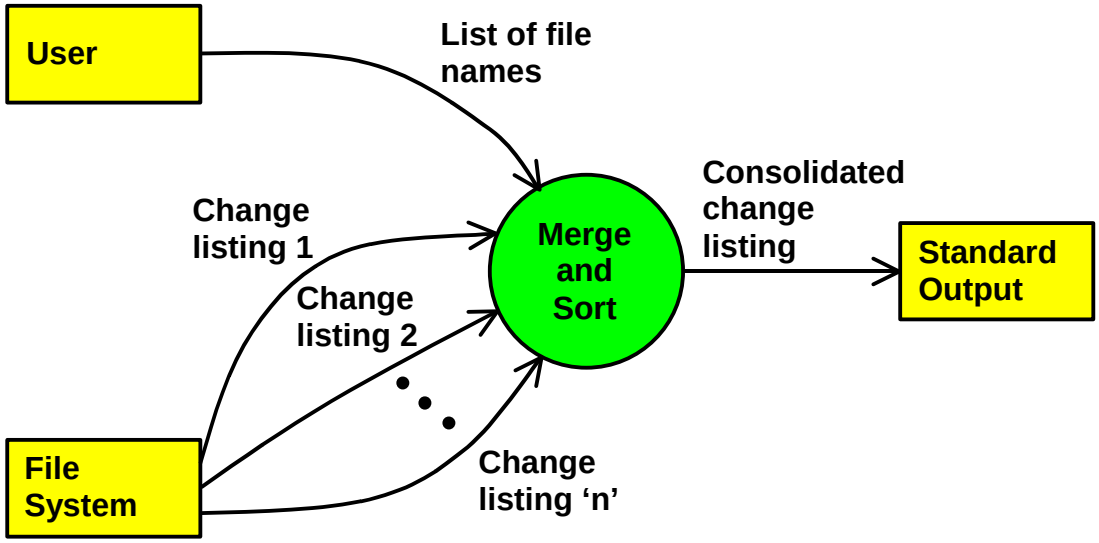
Each change listing shows the changes to multiple source files.

In this case, the change listing contains changes to two source files, GuiComp and GuiIncd.

The change listing shows the changes that were made to each source file. In this case, both programs were changed to show ‘Shift date’ instead of ‘Comp issue loc date’.

A header line that starts with a row of equals-signs identifies each file that was changed. The header includes a file name, a release, in this case 37 and a level, in this case 2.

3.2 External Data Flows



The question paper tells us that the merge and sort program must accept a list of files from the command line, each containing a change listing of the type shown on the previous slide, sort the changes in the change listings by file name and then version and then emit the changes, in the same format as the input change listings to the program's standard output.

So here is a simple level-one dataflow diagram of the merge and sort program.

It must accept a list of file names from the user.

It must read the data in the files from the file system.

It must consolidate the data in the files and sort it by file name and version and then emit the consolidated changes as a single output stream, in the same format as the input files, to standard output.

3.3 Record Structure

```
struct Change_t {
    string      changeName;
    int         changeRelease;
    int         changeLevel;
    string      changeData;
};
```

To sort and merge the data about individual changes in the change listings, we first of all need to put the data concerning each individual change into a record that can be moved around as if it were a single item.

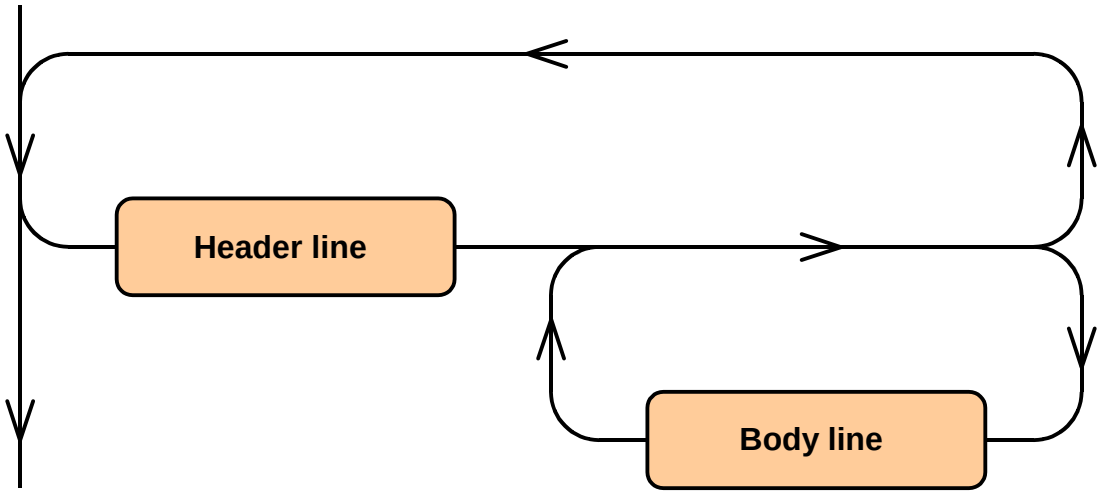
In C++, the record is an instance of a structure. This slide shows the type of structure that could be used to store the data about a change.

The first three elements in the structure are the keys that are needed for sorting the records.

The file name can be stored in a C++ string and the release and level numbers can be stored in integers.

The last element in the structure is the change data that must be emitted to standard output after the changes have been sorted.

3.4 Change Listing Syntax



The problem we now face is to load the update structures from the data in the input files.

When solving this problem, it can be a big help to sketch a syntax diagram that describes the input syntax.

In this case each change listing consists of header lines and body lines. The question paper tells us that a header line begins with an equals sign and a body lines never begins with an equals sign. This provides us with an easy means to identify header and body lines.

The syntax of a change listing is that it typically begins with a header line, followed by zero or more body lines, followed by another header line or the end of the file.

Translating the syntax diagram into a program that reads the input is quite easy. The lines of control in the program will generally follow the lines of control in the syntax diagram.

3.5 Reading an Input File

```
ifstream    ifs;    // Input file stream
string      line;   // Input line
ifs.open (fileName);
getline (ifs, line);
while ( ! ifs.eof()) {
    if (line[0] != '=') {
        cerr << "Error: first line not header\n";
        exit (1);
    }
    //...
    // Process header line
    //...
    getline (ifs, line);
    while ( ! ifs.eof() && line[0] != '=') {
        // Process body line
        getline (ifs, line);
    }
}
ifs.close ();
```

Looking at the code for reading the lines in the input file, we can see that each loop in the syntax diagram corresponds to a loop in the program that reads the file.

To process the file, the program opens the file and reads a look-ahead line. A look-ahead line, token or character is an important feature of any program that validates and interprets an input syntax.

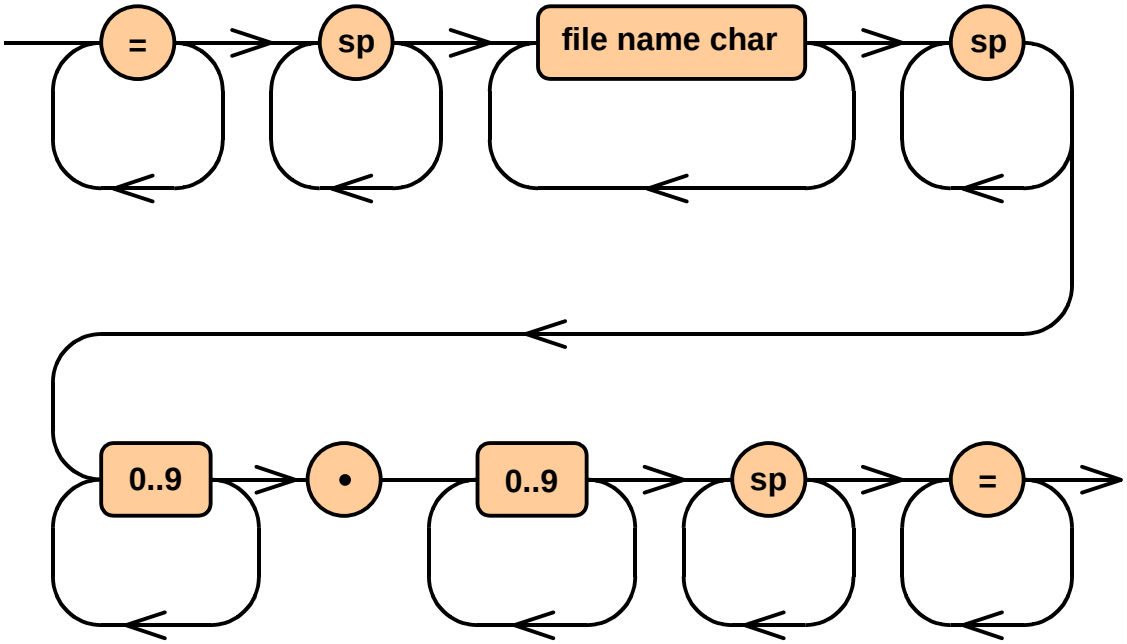
Each section of the input file must start with a header line, which is identified by an equals sign. If the first line does not start with an equals sign, it is an error.

After successfully reading a header line, the program can process the header line and then advance the look-ahead line forward to the next line.

If the next line is not end-of-file and is not a header line, it must be a body line. When the program encounters a body line, it can process the body line. In this case processing a body line will involve storing the body line in the updateData member of the Update_t structure. When it has processed the body line, the program once again advances the look-ahead line and goes back to see whether the look-ahead line is a header line, a body line or the end of the file.

Because the logic of the program that processes the input follows the sequence of the syntax diagram, it is often useful to draw a syntax diagram to assist in figuring out how to create a program to read the input.

3.6 Header Line Syntax



Now we are faced with the problem of extracting the file name, release and level from the header line, and we do the same thing.

Start by drawing a syntax diagram.

The header line starts with a string of equals signs, followed by a space. I would generally allow one or more spaces. This is followed by the file name, which the question paper states does not contain embedded spaces, and another space, and again, I would generally allow one or more spaces.

After the file name is a sequence of digits for the release number number, followed by a decimal point and another string of digits for the level number. The line finishes with one or more spaces followed by one or more equals signs.

3.7 Extracting the file name

```
i = 0;
ch = line[i++];
if (ch != '=') throw error;
do ch = line[i++]; while (ch == '=');
if (ch != ' ') throw error;
do ch = line[i++]; while (ch == ' ');
fileName = "";
while (ch != ' ' && ch != '\0') {
    fileName += ch;
    ch = line[i++];
}
if (ch != ' ') throw error;
do ch = line[i++]; while (ch == ' ');
```

To extract the file name from the header line, again we make the program follow the input sequence as defined in the syntax diagram. Where we have a loop in the syntax diagram, we have a loop in the program.

The program starts by loading the look-ahead character with the first character in the line.

The program then checks that the first character is an equals sign. If not, it throws an error that can be caught and processed by an outer layer. Then the program skips over the string of equals signs.

The program then checks that the next character is a space. If not, it throws an error. If the space is found, the program skips over one or more spaces.

Next, the program reads the file name. It starts by resetting the file name string. It then reads file name characters, appending them to the file name variable until either a space character or an end-of-string character is found.

If the file name does not end in a space, the program throws an error. Otherwise it skips over the space characters.

Again, notice that the logic of the program follows the sequence of the syntax in the syntax diagram.

3.8 Extracting the Release and Level

```
if ( ! isdigit(ch)) throw error;
release = 0;
do {
    release = release*10 + ch - '0';
    ch = line[i++];
} while (isdigit(ch));
if (ch != '.') throw error;
ch = line[i++];
if ( ! isdigit(ch)) throw error;
level = 0;
do {
    level = level*10 + ch - '0';
    ch = line[i++];
} while (isdigit(ch));
if (ch != ' ') throw error;
do ch = line[i++]; while (ch == ' ');
if (ch != '=') throw error;
do ch = line[i++]; while (ch == '=');
if (ch != '\0') throw error;
```

Extracting the release and level from the header line is much the same as extracting the file name.

In this case, we check that the release starts with a digit. Next, the program loops through release digits, converting each digit to its binary equivalent and appending the binary digit to the release variable.

After reading the release, the program checks that the next character is a decimal point, advances the look-ahead character and then reads the level number in the same way as the release number.

At the end, the program can check that the header line finishes with a space and another string of equals signs.

3.9 Loading a Vector of Changes

```
vector<Change_t> ChangeVec_t;
Change_t         change;
ChangeVec_t      changeVec;

// To append a new change record to
// the end of the vector:

changeVec.push_back (change);
```

In the sample solution, as the individual changes are extracted from the input files, they are appended to a change vector.

Can anyone distinguish a vector from an array?

Once all the changes have been appended to the vector, we can use the C++ sort algorithm to sort the changes by file name, release and level, but first of all we need a way of telling the sort algorithm the order in which the changes are to be sorted.

3.10 Specifying the Sorting Order

```
struct Change_t {
    string      changeName;     // Changed file name
    int         changeRelease;  // Release number
    int         changeLevel;    // Level number
    string      changeData;     // Change data

    // Less-Than Operator

    bool
    operator < (
        const Change_t &change) // Value to compare
    const
    {
        if (changeName < change.changeName) return 1;
        if (changeName > change.changeName) return 0;
        if (changeRelease < change.changeRelease) return 1;
        if (changeRelease > change.changeRelease) return 0;
        return changeLevel < change.changeLevel;
    }
};
```

The easiest way to specify a sorting order in C++ is to provide a less-than operator for comparing the records to be sorted. This is done by specifying a less than operator as a member function in the structure declaration.

In this example, the less than operator first compares the records by the file name. If the file names are the same, it compares the records by release and then if the releases are the same, it compares the levels.

3.11 Sorting the Vector

```
sort (changeVec.begin(), changeVec.end())
```

Having set up a less than operator, invoking the C++ sort algorithm is trivial.

All that remains after sorting the records is to emit change records in the order they are stored in the change vector, which is also trivial.

3.12 Summary

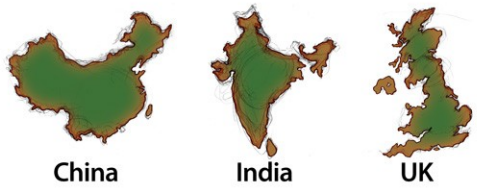
The Merge and Sort problem was a very easy problem. This is all you needed to know to answer the question:

- How to get access to command line parameters from a program.
- How to open and read characters from a text file.
- How to read and interpret input sequences.
- How to use the language or operating system sort functions.
- How to emit text to the program's standard output.

These are basic skills.

4 FINANCIAL STATEMENTS

4.1 Introduction



The Financial Statements problem is a reasonably straightforward problem in generating a report from an SQL database.

4.2 Sample Report

```
22-09-07 11:23     FINANCIAL STATEMENTS              PAGE   1
                      as at 31-08-07

Line
No     Description                                     Balance

0100   Helical Lamps on Hand                          58,267.20
0110   Spiral Lamps on Hand                          142,816.85
0120   Flamingo Lamp Shades on Hand                    5,628.50
0130   Assorted Electrical Fittings on Hand            1,252.10
0180                                                 ----------
0190   Total value of stock on hand                  207,964.65
0200
0210   Cash at Bank, Antopolis Commercial Bank        22,865.12
0220   Cash at Bank, Bank of Trioville                71,102.78
0280                                                 ----------
0290   Total cash at bank                             93,967.90
0300
0310   Trade creditors                               102,781.25
0320   Mortgage on warehouse                          50,215.12
0380                                                 ----------
0390   Total liabilities                             152,996.37
0400                                                 ----------
0410   Net worth                                     148,936.18
0420                                                 ==========
```

The question paper gives us this sample of the report that is required.

The data on the database must be backdated to a date provided by the users. I.e. the report is to be produced 'as at' some particular date.

The left hand column is a line number that relates to line number stored on the database.

The middle column of the report contains descriptions of the figures shown on the right hand column of the report.

There are four different types of line:

1. There are balance lines, which display a figure in the right column;
2. There are blank lines;
3. There are single-underline lines; and
4. There are double-underline lines.

The data in the database can be divided into two groups of tables. One group of tables, we will call them the report definition tables, defines the lines to be shown on the report; and the other group of tables, the transaction tables, contain the financial data to be analysed to produce the numbers shown on the right hand side of the report.

4.3 Report Definition Tables

```
create table lines (
     lineNo         smallint not null,
     lineDescr      char(40) not null,
     lineType       smallint not null
);
create table elements (
     eleLineNo      smallint not null,
     eleSign        smallint not null,
     eleAccId       char(10) not null
);
create table references (
     refThisLineNo  smallint not null,
     refSign        smallint not null,
     refOtherLineNo smallint not null
);
```

There are three report definition tables.

The first is the lines table. The question paper tells us that the lines table contains one row for each line on the financial statements. The line number is the figure to be shown in the left column of the financial statements; the line description is the string to be shown in the centre column of the financial statements; and the line type is a code, either 0, 1, 2, or 3 for balance lines, blank lines, single underlines or double underlines respectively.

The elements and references tables define the account balances to be included in the balance to be shown in the right hand column of a given line number. The two tables allow the balance to be derived from arithmetic combinations of account balances and other lines on the report.

Each row in the elements table identifies an account balance to be included in the line balance to be shown on the report and each row in the references table identifies another line balance to be included in the line balance. The sign columns are either +1 or –1 depending on whether the account balance or other line balance is to be added to the line balance or subtracted from the line balance respectively.

4.4 Formula for Line Balance

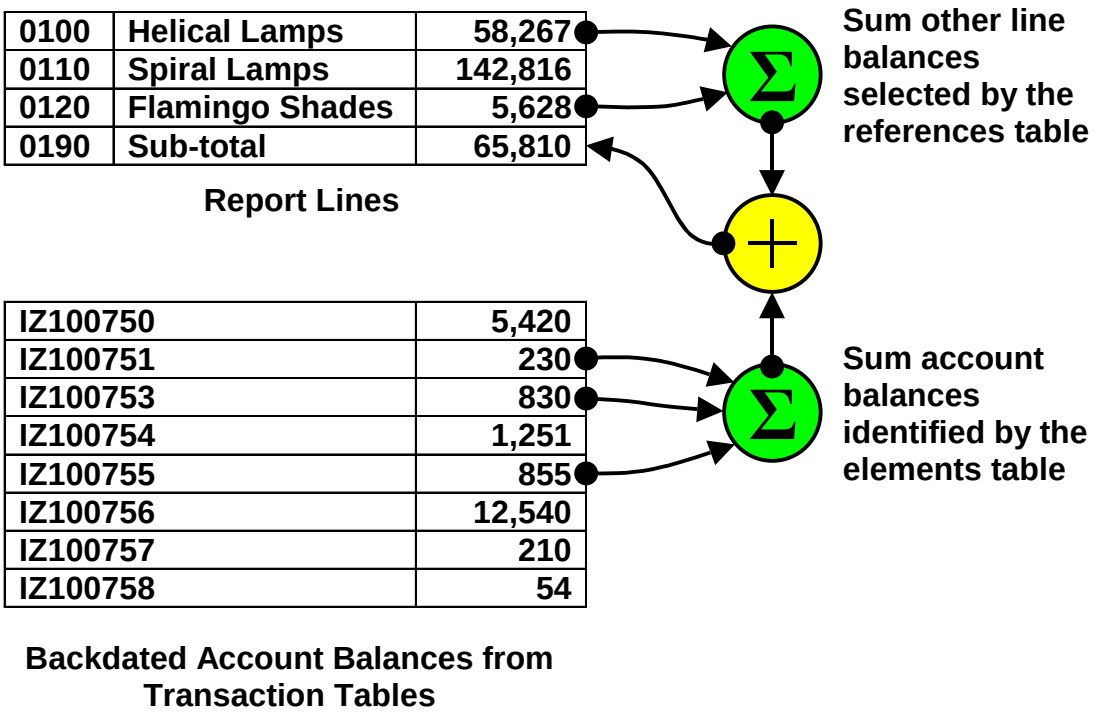
$$\text{lineBal}_i = \sum_{j=1}^{n} \text{eleSign}_j * \text{backBal}_j + \sum_{k=1}^{m} \text{refSign}_k * \text{lineBal}_k$$

Where:

lineBal_i	is the line balance of the i-th line.
n	is the number of `elements` table rows with `eleLineNo` equal to the line number of the i-th line.
eleSign_j	is the value of `eleSign` in the j-th `elements` table row with `eleLineNo` equal to the line number of the i-th line.
backBal_j	is the back-dated balance of the account identified by `eleAccountId` in the j-th `elements` table row with eleLineNo equal to the line number of the i-th line.
m	is the number of `references` table rows with a `refLineNo` equal to the line number of the i-th line.
refSign_k	is the value of `refSign` the k-th `references` table row with refThisLineNo equal to the line number of the i-th line.
lineBal_k	is the line balance of the other line identified by `refOtherLineNo` in the k-th references table row with `refThisLineNo` equal to the line number of the i-th line.

This was the formal definition of a line balance given in the question paper. It looks like a bit of a monster, but it's really not that complicated.

4.5 Calculating Line Balances



This slide is a pictorial representation of the same thing.

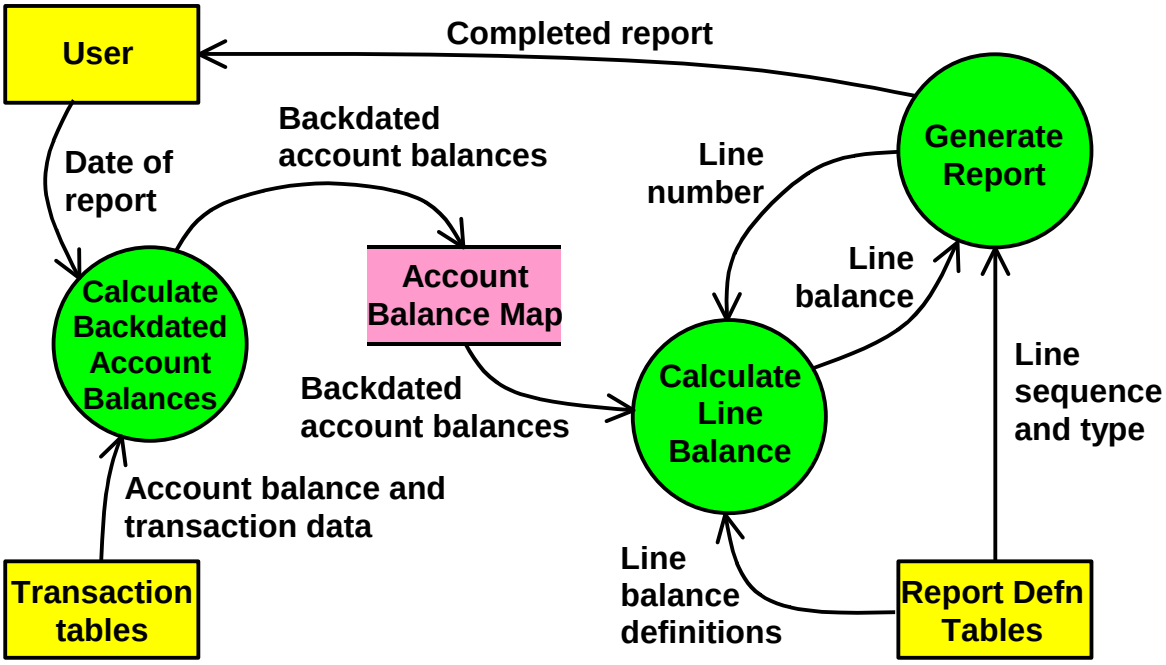
The elements table identifies the backdated account balances to be included in the line balance.

The references table identifies the line balances of other lines to be included in the line balance of this line.

The sign columns in the elements and references table, which are not shown on this diagram, indicate whether the component is to be added to or subtracted from the total.

From this diagram it is easy to see how the calculation of a line balance is naturally recursive. The line balance of one line can become a component of the line balance of another line.

4.6 Dataflow Diagram



Putting all this together into a dataflow diagram, we can see what is needed to program the report.

First, we need to deal with the business of calculating the account balances as at the date of the report. These can be stored in a memory-resident data structure to make it readily accessible to the function that calculates line balances.

Next the module that generates the report can loop through the definition of each line, which it can extract directly from the database. When it needs a line balance, it can call another module to total up the backdated account balances and other line balances.

4.7 Transaction Tables

```
create table accounts (
     accId          char(10) not null,
     accDescr       char(60) not null,
     accBalance     double precision not null
);
create table transactions (
     txDate         date not null,
     txNo           integer not null,
     txDescr        char(60) not null,
     txFromAccId    char(10) not null,
     txToAccId      char(10) not null,
     txValue        double precision not null
);
```

Let us now look at the source of the transaction data to see how the backdated balances can be calculated.

The question paper supplied this definition of the transaction tables.

The accounts table has one row for each account. It contains an account identifier, an account description and the current balance of the account.

The transactions table has one row for each transaction. It contains a transaction date, a transaction number, which is a number that uniquely identifies each transaction, a transaction description, a 'from' account identifier, a 'to' account identifier and a transaction value.

The 'from' and 'to' account identifiers identify the accounts between which the transaction value moved. They identify the accounts that were credited and debited respectively.

4.8 Definition of the Account Balance Map

```
// ACCOUNT BALANCE MAP

typedef map<string,double> AccBalMap_t;
```

In order to use the data in the transaction tables, the program that generates the financial statements must backdate the account balances stored in the accounts table to the date of the report.

In the sample solution, this was done by back-dating the balances in an account balance map.

This is how the account balance map was declared. It makes use of the C++ map template.

4.9 Loading the Account Balance Map

```
// Load the current account balances into the account
// balance map

accBalMap.clear();
exec sql declare accCur cursor for
        select  accId, accBalance
        from    accounts;
exec sql open accCur;
for (;;) {
        exec sql fetch  accCur
                into    :accId, :balance;
        if (SQLCODE != 0) break;
        accBalMap[accId] = balance;
}
exec sql close accCur;
```

Loading the account balances into the map is a simple matter of using an SQL cursor to read the account balances from the database and load them into the account balance map.

4.10 Backdating the Account Balances

```
// Roll back the effect of the transactions between the
// statement date and now

exec sql declare txCur cursor for
        select  txFromAccId, txToAccId, txValue
        from    transactions
        where   txDate > :dbDate;
exec sql open txCur;
for (;;) {
        exec sql fetch  txCur
                into    :fromAccId, :toAccId, :value;
        if (SQLCODE != 0) break;
        accBalMap[fromAccId] += value;
        accBalMap[toAccId] -= value;
}
exec sql close txCur;
```

Calculating the account balances as at the date of the report is a simple matter of reversing the effect of the transactions that were processed after the date of the report.

This cursor declaration selects all the transactions after the date of the report.

The transaction value can then be added back to the account from which it was originally deducted and deducted from the account to which it was originally added.

It really is quite straightforward.

4.11 Calculating a Line Balance

```
CalcLineBal:
     lineBal = 0;
     For each row in elements where eleLineNo
     equals the required line number:
          lineBal += eleSign * accBalMap[eleAccId];
     For each row in references where refThisLineNo
     equals the required line number:
          Recursively call CalcLineBal
          to calculate the line balance of the
          refOtherLineNo;
          lineBal += refSign *
               line balance of refOtherLineNo;
     Return lineBal.
```

This is how to calculate a line balance in pseudo-code.

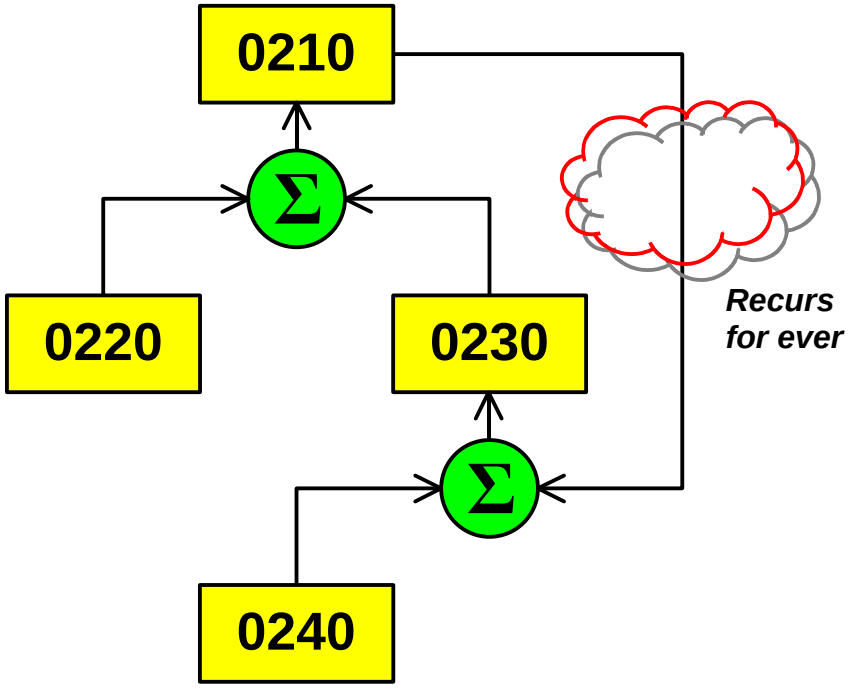
First the function must reset a line balance accumulator.

Then the function adds the product of the sign column and the account balance map for each account balance to be included in the line.

Next, for each other line balance to be included in this line balance, the function recursively calls itself to calculate the line balance of the other line and then multiplies the balance by the sign and adds it to the line balance accumulator.

When all the components of the line balance have been calculated, the function returns the calculated line balance.

4.12 What are Self-Referencing References?



The question paper tells us that the program should endeavour to identify self-referencing references table entries and gives us the example shown on this slide.

In this case when the program attempts to evaluate the balance to be shown on line 0210, it must add up the balances of lines 0220 and 0230. No problem there, but when it comes to evaluate the balance of line 0230, it finds that it is the sum of lines 0240 and 0210, at which point it recurs to try and calculate the balance of line 0210. Indeed it will continue to recur indefinitely until the program runs out of memory.

4.13 Detecting Self-Referencing References

```
CalcLineBal:
     lineBal = 0;
     Lock the line;
     For each row in elements where eleLineNo
     equals the required line number:
          lineBal += eleSign * accBalMap[eleAccId];
     For each row in references where refThisLineNo
     equals the required line number:
          If refOtherLineNo is locked:
               Reject: self-referencing reference;
          Recursively call CalcLineBal
          to calculate the line balance of the
          refOtherLineNo;
          lineBal += refSign *
               line balance of refOtherLineNo;
     Unlock the line;
     Return lineBal.
```

Detecting the self-referencing references is simply a matter of locking each line while the balance to be shown on the line is displayed. If the new line to be evaluated is locked when the program recurs, then it must be a self-referencing recursion, in which case some kind of rejection can be thrown.

In the sample solution, the locking mechanism was a C++ set that contained the line numbers of the locked lines.

4.14 Report Layout

```
         1         2         3         4         5         6
1...5....0....5....0....5....0....5....0....5....0....5....0
DD-MM-YY HH:MM     FINANCIAL STATEMENTS              PAGE X-X
                      as at XX-XX-XX

Line
No     Description                                     Balance

XXXX   X---------------------------------------X  XXX,XXX.XX
XXXX   X---------------------------------------X  XXX,XXX.XX
XXXX   X---------------------------------------X  ----------
XXXX   X---------------------------------------X  XXX,XXX.XX
XXXX   X---------------------------------------X  ==========
```

Each year we provide a report layout for the reporting program question and each year it is all but ignored.

The bottom line is that we do mark contestants down for failing to lay the report out in accordance with the report layout.

The report layout should be interpreted in this way:

1. the date and time the report was generated should be shown in the top left of each page;
2. the report should carry the title 'FINANCIAL STATEMENTS';
3. the page number of each report page should be shown on the top right corner.
4. the report should show the date of the report below the title;
5. the line number should be shown in a 4-digit field starting in column one;
6. the line description should be shown in a 40-character field starting at column 8;
7. the line balance should be shown in a 10-character field starting at column 50 and should include comma separators and two decimal places.

If you write in C or C++, the functions for formatting titles, dates and dollar values could simply have been taken from the answer to last year's paper. Though if you did so, we would appreciate a comment acknowledging the source.

4.15 Generating the Report

```
for (;;) {
    exec sql fetch  lineCur
             into    :lineNo, :lineDescr, :lineType;
    if (SQLCODE != 0) break;
    cout << setfill('0') << setw(4) << lineNo << "   ";
    cout << setfill(' ') << left << setw(42) << lineDescr;
    switch (lineType) {
    case LINE_TYPE_BALANCE:
        cout << right << setw(10) << FormatNum (
                CalcLineBal (lineNo, &lockSet));
        break;
    case LINE_TYPE_BLANK:
        break;
    case LINE_TYPE_SINGLE:
        cout << "----------";
        break;
    case LINE_TYPE_DOUBLE:
        cout << "==========";
        break;
    }
    cout << '\n';
}
```

Actually emitting the report lines is simply a matter of executing this sequence for each line in the lines table.

First the program reads the line number, line description and line type from the lines table.

It then emits the content common to all line types, the line number and line description.

Next it switches depending on the line type.

If the line is a balance line, it uses the recursive CalcLineBal function to calculate the line balance to be shown on the report.

If the line is a blank line, it emits nothing.

If the line is a single or double underline, it emits the row of dashes or equals signs.

Lastly, it emits the end-of-line character.

And that's all there is to this problem.

4.16 Summary

The Financial Statements question was yet another problem in extracting data from an SQL database and presenting the data in a report to be presented to users. The creation of these kinds of programs is what feeds the software development industry. Every year every respectable business needs a dozen or so of these types of reports written or amended in some way. It is something you need to learn how to do.

This is what you could learn how to do by solving the Financial Statements problem:

- Interpreting reporting program specifications.
- Interpreting equations in program specifications.
- Understanding a report layout.
- How to create a program that extracts data from an SQL database.
- Using dataflow analysis to break a big problem down into manageable components.
- Using language data structures such as maps and sets to store intermediate values.
- Using recursion to evaluate nested expressions.
- Checking for self-referencing references.
- Using formatting functions to generate a report.

5 THE BANKCARD KID

5.1 Introduction

The BankCard Kid

He's Got the Fastest Plastic in the West

The question paper tells us that a British entrepreneur by the name of Peter Burns has set up a low-cost banking business, but the touch-screen ATMs he had programmed by an overseas software house, occasionally and unpredictably throw up a message that says 'atmruntime.exe has encountered a problem and needs to close. We are sorry for the inconvenience…'.

Customers are figuring they've been burnt again and the whole thing is about to go down the gurgler.

Burns' programmers are having trouble locating the cause of the problem because it is very hard to trigger in a testing environment. To help identify the cause of the problem they would like to have a program they have dubbed the 'Bankcard Kid' that repeatedly submits random transactions to the ATM. Our task is to program the Bankcard Kid.

5.2 Testing Cycle

1. repeat until terminated:
    a. randomly select a card from the available alternatives;
    b. emulate the insertion of the card into the ATM;
    c. wait for between 3 and 7 seconds;
    d. check that the card was accepted;
    e. emulate the entry of a PIN, waiting for between 0.05 and 1.0 seconds between each emulated touch of the screen;
    f. wait for between 1.5 and 3 seconds;
    g. check that the PIN was accepted;
    h. pseudo-randomly select an account (cheque, savings or credit);
    i. wait for between 1.5 and 3 seconds;
    j. check that the account selection was accepted;
    k. emulate entry of an amount to be withdrawn, waiting for between 0.05 and 1.0 seconds between each emulated touch of the screen;
    l. wait for 5.0 seconds;
    m. check that the correct amount is paid out;

The question paper tells us that the Bankcard Kid must execute this cycle.

5.3 Programming Interface

```
public class ATMInfra {
          static public void emulateCardIn (
               String cardNumber) {/*...*/}
                         // Emulate card insertion
          static public int [] readScreenPixels ()
               {/*...*/}
                         // Read pixels from screen
          static public int [] readScreenDump (
               const char *fileName) {/*...*/}
                         // Read pixels from screen dump
          static public void emulateTouch (
               int x, int y) {/*...*/}
                         // Emulate screen touch
          static public int getAmountPaid ()
               {/*...*/}
                         // Get the amount paid
          //...
     }
```

The question paper also tells us about an interface to the ATM infrastructure and device drivers that let us perform the functions needed to exercise the ATM.

emulateCardIn emulates the insertion of a card into the card reader.

readScreenPixels reads the RGB colour components of the pixels on the screen. It is needed to determine whether an operation has succeeded or failed.

readScreenDump reads RGB colour components stored in a disk file. Screen dumps from previous, successful operations are stored in disk files. To test whether an ATM operation has succeeded or failed, the Bankcard Kid must compare the current screen image with a screen image from a previous successful run stored on a disk file.

emulateTouch emulates a user touching the touch-screen of the ATM. The Bankcard Kid can use emulateTouch to select options and key in numeric values.

getAmountPaid returns the amount paid out by the ATM since getAmountPaid was last called. When the Bankcard Kid starts up, it must call getAmountPaid to reset the internal counter of the amount paid out. It can then call getAmountPaid to see if the correct amount has been paid out.

5.4 Test Cards

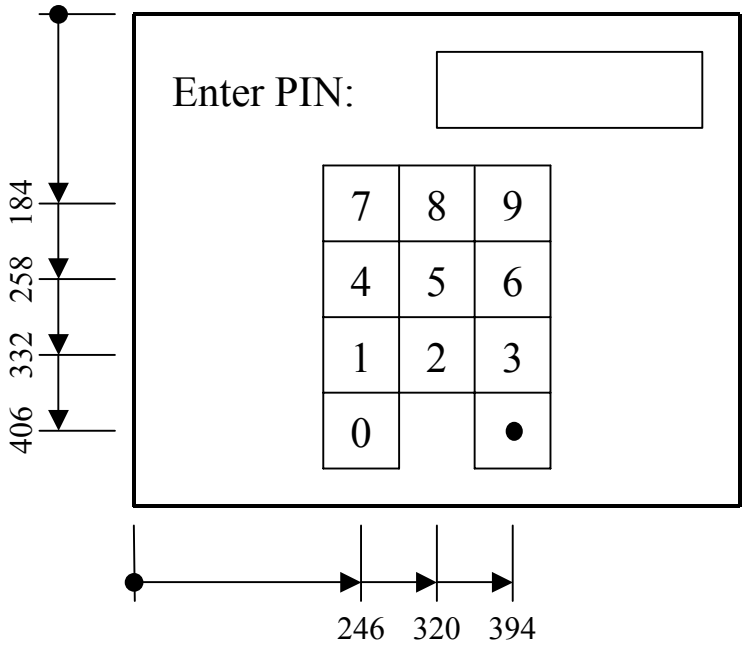
```
0856669126311954 212384
4996552050476648 806909
9764925646670808 089694
7570032704671097 693010
1516410663483838 565200
6611851186922681 140063
0969351051899105 097351
4928296487649896 272250
6505872582625350 543119
2906025111421829 668369
       :           :
```

The first step in the testing cycle is to randomly select a card from the available alternatives.

The available alternatives are stored in a file called ‘TestCards.txt’. The file contains approximately 50 card number-PIN pairs.

This slide is the first 10 lines of TestCards.txt. The first 16 characters are a card number and then a single space and the corresponding 6-digit PIN. Each card number-PIN pair is listed on a separate line.

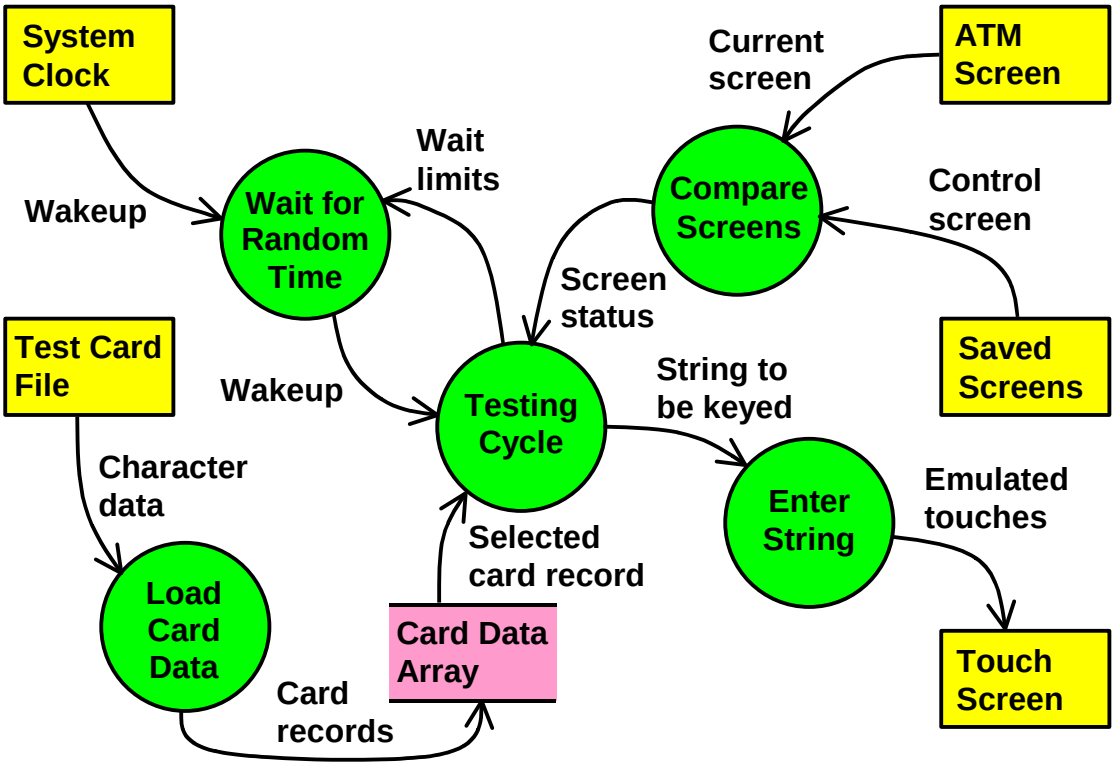
5.5 Screen Layout



The question paper includes a number of screen layouts such as this one for entering the customer's PIN.

The screen layouts give us the co-ordinates of the points on the screen that must be touched to activate functions and enter numbers.

5.6 Dataflow Diagram



Writing a program to execute the testing cycle is all very well, but before we do that we must think a little about the support functions that the testing cycle might need.

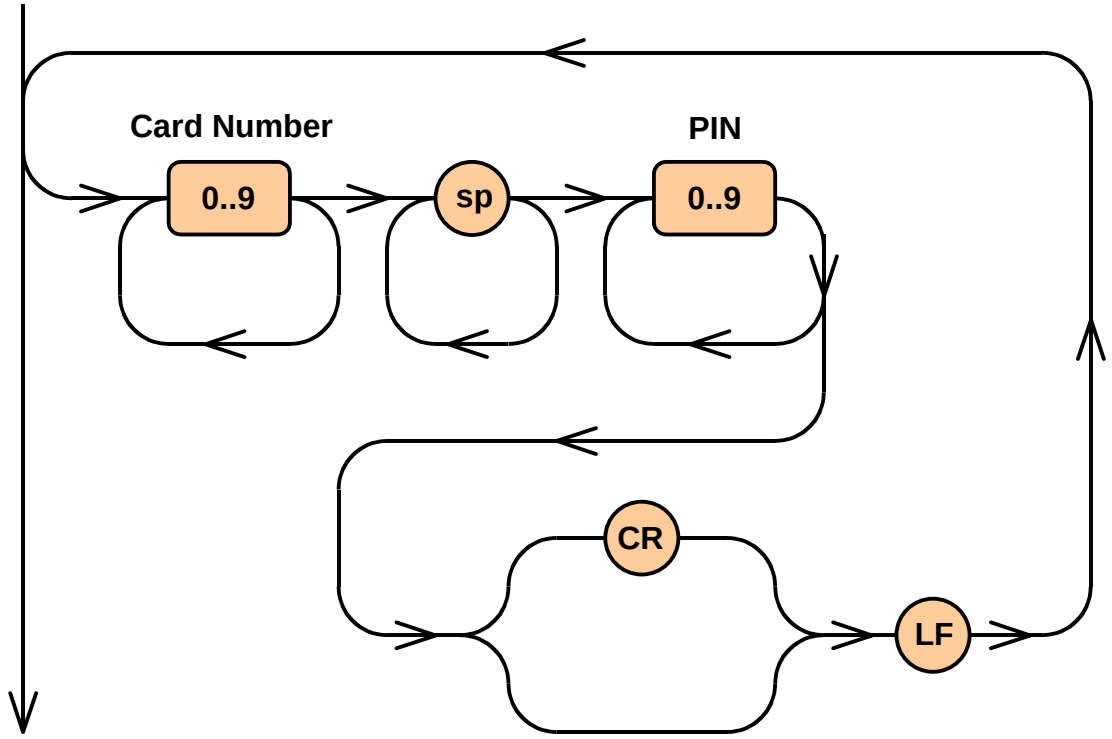
First, we must read the card number-PIN pairs into an array so that the testing cycle can select a random card number-PIN pair by generating a random array index.

We are going to need a means for entering numeric values into the touch screen. It will be a lot easier to program the testing cycle if the testing cycle can call a function to key in $250, rather than having to look up the co-ordinates for the digit 2 and call emulateTouch, and then look up the co-ordinates for the digit 5 and call emulateTouch and then look up the screen co-ordinates for the digit zero and call emulateTouch and so on for the decimal point and two extra zero digits.

It would also be handy if the testing cycle could just all a function to validate the image displayed on the screen. The question paper tells us that because of a flashing cursor, the screens do not match exactly, they may have up to 200 differing pixels. If this complication is dealt with by a function, the testing cycle will be simpler.

Similarly, the testing cycle has a frequent need to wait for a pseudo-randomly selected time between two limits. If the testing cycle can call a function to select the wait period and do the wait, it will be easier to program.

5.7 Card Data Syntax



To read the input file, again it is a good idea to draw a syntax diagram of the input sequence. This is the same underlying process that we used to read the input data for the merge and sort problem.

In this case, the file starts with a sequence of digits, and then a space. I would generally allow one or more spaces to make the program more flexible. Next there is a sequence of digits for the PIN. The line is finished by a new line sequence, which might be either a carriage return, line feed pair or just a line feed character.

The line sequence is repeated in the card data file until it is terminated by end-of-file.

5.8 Card Data Reader

```
// Open the file and load the look-ahead character

fileReader = new FileReader (CARD_DATA_FILE_NAME);
ch = fileReader.read();

// Read card data lines until end-of-file is encountered

while (ch != -1) {
    cardData = new CardData();

    // Read the card number

    if ( ! Character.isDigit(ch))
        throw new RuntimeException ("bad char");
    stringBuffer.setLength (0);
    do {
        stringBuffer.append ((char)ch);
        ch = fileReader.read();
    } while (Character.isDigit(ch));
    cardData.cardNumber = stringBuffer.toString();
```

Again to read the input file we load a look-ahead character, in this case the variable 'ch', and then make the sequence of the program follow the input syntax.

There is an outer while loop in the input syntax that defines input lines. There is an outer while loop in the reader that decides whether a new input line needs to be read. In Java, the character value –1 indicates end-of-file.

To read the card number, we first check that the first character of the card number is a digit. Looking back at the syntax diagram, we can see that the first character in each line must be a digit. If any other character is encountered, the input file is invalid.

Then again, there is a do-until loop in the input syntax that is terminated by a non-numeric character, consequently there is a do-until loop in the reader that exits when a non-numeric character is encountered.

The remaining fields in the input file can be processed in the same way.

I cannot overstress the usefulness of this simple parsing technique. Get a look-ahead character and then let the program follow the sequence of the input syntax deciding what to do next based on the value of the look-ahead character. This simple technique can read simple sequences such as this and merge and sort input file, right up to complex syntaxes such as those of the Java and C++ languages. To paraphrase Mark Twain, it's a skill worth your learning.

5.9 Using Vectors

```
vector<CardData> cardDataVec;
cardDataVec = new Vector<CardData> (50, 10);
// ...
while (ch != -1) {
    cardData = new CardData();
    // ...
    // Load cardData from the input file
    // ...
    cardDataVec.add (cardData);
}
return cardDataVec.toArray(new CardData(1));
```

The question paper tells us that the card data file contains approximately 50 sample card numbers and PINs, but we do not know exactly how many sample cards there are.

This is an application for a vector in both Java and C++. A vector is like a variable length array.

In Java, we create a vector by declaring the vector and the type of the vector element. The vector constructor has parameters for the initial capacity of the vector and the extent allocation size. This means that the Java virtual machine will initially allocate space for 50 card numbers and then if 50 is not enough, it will allocate space for another 10 and so on.

To append an element to the vector, the program calls the add method.

At the end of the module that loads the card data in the sample solution, the program converts the vector into an array because in Java, arrays are much easier to use than vectors.

5.10 Selecting a Pseudo-Random Card Number

```
// Randomly select a card from the available alternatives

cardInd = randGen.nextInt (cardDataArr.length);

// Emulate insertion of the card into the ATM

ATMInfra.emulateCardIn (cardDataArr[cardInd].cardNumber);
```

Pseudo-randomly selecting a card from the available alternatives is quite easy.

In the sample solution, randGen is an instance of the Java intrinsic class Random. The nextInt method returns a pseudo-random integer between zero and the parameter minus one. This value can be directly used as an index to an array.

In this case the index is used to select the card number from the card data array and pass the class number to the emulateCardIn interface function.

5.11 Wait for a Pseudo-Random Period

```
static private void
randWait (
    int     minWait,  // Minimum waiting period (ms)
    int     maxWait)  // Maximum waiting period (ms)
{
    int     period;   // Period to sleep for

    try {
        period = minWait +
            randGen.nextInt (maxWait - minWait + 1);
        Thread.sleep (period);
    }
    catch (InterruptedException e) {
        throw new RuntimeException (e.toString());
    }
}
```

Waiting for a pseudo-random period is a recurring requirement of the Bankcard Kid program. Consequently, it is worth writing a function to perform this function.

In the sample solution there is a static function, randWait, that accepts two parameters, a minimum waiting period in milliseconds and a maximum waiting period in milliseconds.

randWait uses an instance, randGen, of the Java random number generator, Random, to select a pseudo-random number between zero and the range of the random variation. It then adds back the minimum waiting period to obtain a pseudo-random period between the two limits.

It then uses the Java sleep function to cease execution for the generated period.

This function can be used throughout the Bankcard Kid program to wait for a pseudo-random period when this is required.

5.12 Entering Strings into the Touch Screen

```
static private void
enterString (
    String      str)    // String to be keyed
{
    int         i;      // General purpose index

    for (i = 0; i < str.length(); i++) {
        if (i != 0) randWait (50, 1000);
        switch (str.charAt(i)) {
        case '0':
            ATMInfra.emulateTouch (246, 406);
            break;
        case '1':
            ATMInfra.emulateTouch (246, 332);
            break;
        // ...
        // Process other characters
        // ...
    }
}
```

Next, we need a function to submit text strings to the touch screen. The testing cycle is going to generate amounts of money of the form '250.00', which need to be converted into emulateTouch calls with the appropriate parameters.

To accomplish this, we can write a function that accepts a string and then converts the characters in the string into the equivalent emulateTouch system calls.

The function loops through the characters in the string. For the second and subsequent character, the function waits for between 50 and 100 milliseconds as required by the question paper.

It then switches on the character to select the co-ordinates to be passed to emulateTouch.

5.13 Comparing Screen Images

```
static private boolean
compareScreen (
    long[]  control)      // Control screen dump
{
    long[]  sample;       // Pixel array from the screen
    int     mismatchCnt;  // Mismatch count
    int     i;            // General purpose index

    sample = ATMInfra.readScreenPixels();
    mismatchCnt = 0;
    for (i = 0; i < 307200 && mismatchCnt <= 200; i++)
        if (sample[i] != control[i]) mismatchCnt ++ ;
    return mismatchCnt <= 200;
}
```

Before we can go to programming the testing cycle, we need one more function. We need a function to compare the display on the screen with the control displays stored on the disk.

In the sample program, the control displays stored on disk were loaded into arrays at the beginning of the program and then the long array was passed to this function.

The function reads the pixels displayed on the screen into to the sample array.

It then counts the number of mismatched pixels. If the number of mismatched pixels exceeds 200, then the screen display is considered to be different to the control dump. If the number of mismatched pixels is less than or equal to 200, then the two images are considered to be equivalent.

The reason the program tolerates mismatched pixels is because a cursor flashes on the screen and depending on whether the cursor is on or off, the image returned by readScreenPixels might be different by up to 200 pixels.

5.14 Programing the Testing Cycle

```
// Enter the PIN

enterString (cardDataArr[cardInd].cardPIN);

// Wait for 1.5 to 3 seconds

randWait (1500, 3000);

// Check that the PIN was accepted

if ( ! compareScreen (acpPINPix)) {
    System.out.println ("Error: PIN for card number " +
        cardDataArr[cardInd].cardNumber +
        " was rejected by the ATM");
    System.exit (1);
}
```

Now we're ready to start programming the testing cycle. This is a program fragment that submits the PIN, waits for the required time and then checks that the PIN was accepted.

The enterString function we devised earlier is used to submit the PIN associated with the randomly selected card.

The program then uses the randWait function to wait for between 1.5 and 3 seconds.

Next it uses the compareScreen function to check that the PIN was accepted. acpPINPix is a long array loaded with the pixels of a screen dump after a PIN is successfully keyed in.

5.15 Checking the Amount Paid

```
// Submit the amount and wait for 5 seconds

enterString (amount + ".00\n");
randWait (5000, 5000);

// Check the amount paid

totAmtPaid = amtPaid = ATMInfra.getAmountPaid ();
while (amtPaid != 0) {
    randWait (1000, 1000);
    amtPaid = ATMInfra.getAmountPaid ();
    totAmtPaid += amtPaid;
}
if (totAmtPaid != amount) {
    System.out.println ("Error: amount mismatch");
    System.exit (1);
}
```

The only other thing that is remotely tricky in programming the Bankcard Kid is counting the amount of money that the ATM paid out.

At the very beginning of the program the application must call getAmountPaid to reset the internal count of the number of bills paid to zero. Every call to getAmountPaid returns the amount paid since the previous call to getAmountPaid.

The question paper specifies that 'after keying in the amount to be withdrawn, the Bankcard Kid must wait five seconds and then call getAmountPaid once per second accumulating the amount dispensed until getAmountPaid returns zero'.

The program waits for 5 seconds by calling randWait with equal parameters of 5000 milliseconds.

It then calls getAmountPaid and initialises the total amount paid with the amount returned by the first call to getAmountPaid.

Next, it checks whether the amount returned by getAmoutPaid is zero, and if it is not, the program waits for a seconds and then calls getAmountPaid again, totalling up the total amount paid until the amount paid in the previous second is zero.

The program then checks that the amount paid matches the amount requested and if not, reports an error and exits.

5.16 Summary

The Bankcard Kid problem was inspired by a real programming problem we had in our office. One of the electronic games we developed was malfunctioning at seemingly random times and we had to create a mad punter to play the game over a long period of time in an effort to find what was causing the problem.

The Bankcard Kid is reasonably straight forward in that it does not involve threads or concurrent processes. The program can execute top to bottom, just calling the operating system's sleep function when needed.

Nevertheless, it is a somewhat more complicated problem than the merge and sort problem. It exercises these skills:

- Identifying useful support functions.
- Reading data from a text file.
- Using vectors.
- Using pseudo-random number generators.
- Sleeping for a pseudo-randomly selected period.
- Converting a string into physical operations.
- Comparing arrays with an error tolerance.
- Using functions that return incremental changes.