E-GENTING PROGRAMMING COMPETITION 2006

PUBLIC LECTURE

3 FEBRUARY 2007

LECTURE NOTES

Second draft
Jonathan Searcy
1 February 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.	**Microcontroller Emulator** A program to emulate a microcontroller with concurrent user input and output functions.	200
2.	**Cashier Activity Report** A commercial reporting program with data accessing and performance complications.	250
3.	**Team Selection** A problem in data analysis and searching.	300
4.	**Loan Evaluator** An almost trivial problem in data entry and computation.	100

In the E-Genting Programming Competition, the hard problems test the students, but the easy ones test the teachers.

Every graduate programmer should be able to answer the Loan Evaluator and Cashier Activity Report questions. They are not hard. They are straightforward run-of-the-mill programming tasks of the kind that every programmer in my office is asked to do several times a year.

Nevertheless, as in past years, an overwhelming majority of the contestants struggled to deliver a credible answer to even the easiest question. So we will start with the easiest question. We will then proceed to look at the Microprocessor Emulator problem, which is a little more interesting. After that we will break for lunch. After lunch we will review the Team Selection problem, which is the most interesting problem, and then finish with the most boring problem, the Cashier Activity Report.

2 LOAN EVALUATOR

2.1 Introduction

Show Loan Evaluator Applet

The Loan Evaluator problem was the result of me trying to come up with the easiest question I could think of that actually did something useful.

The idea was that the program should throw up a data entry screen to receive various parameters about a loan to be borrowed from a bank or a finance company. The problem is that when a bank declares a near-zero interest rate, the interest may well be near zero, but there will usually be bank fees of various kinds that are for all practical purposes, disguised interest. The purpose of the Loan Evaluator is to figure out an effective interest rate for a bank loan given a repayment schedule and the amounts of the fees and other charges.

This is a completed Loan Evaluator. I will show you how it works.

Use the loan evaluator to key in some loan parameters and calculate an effective interest rate.

Show the effect of keying in a non-numeric parameter.

Show the effect of an out-of-range interest rate (e.g. payments less than capital).

2.2 Language Selection

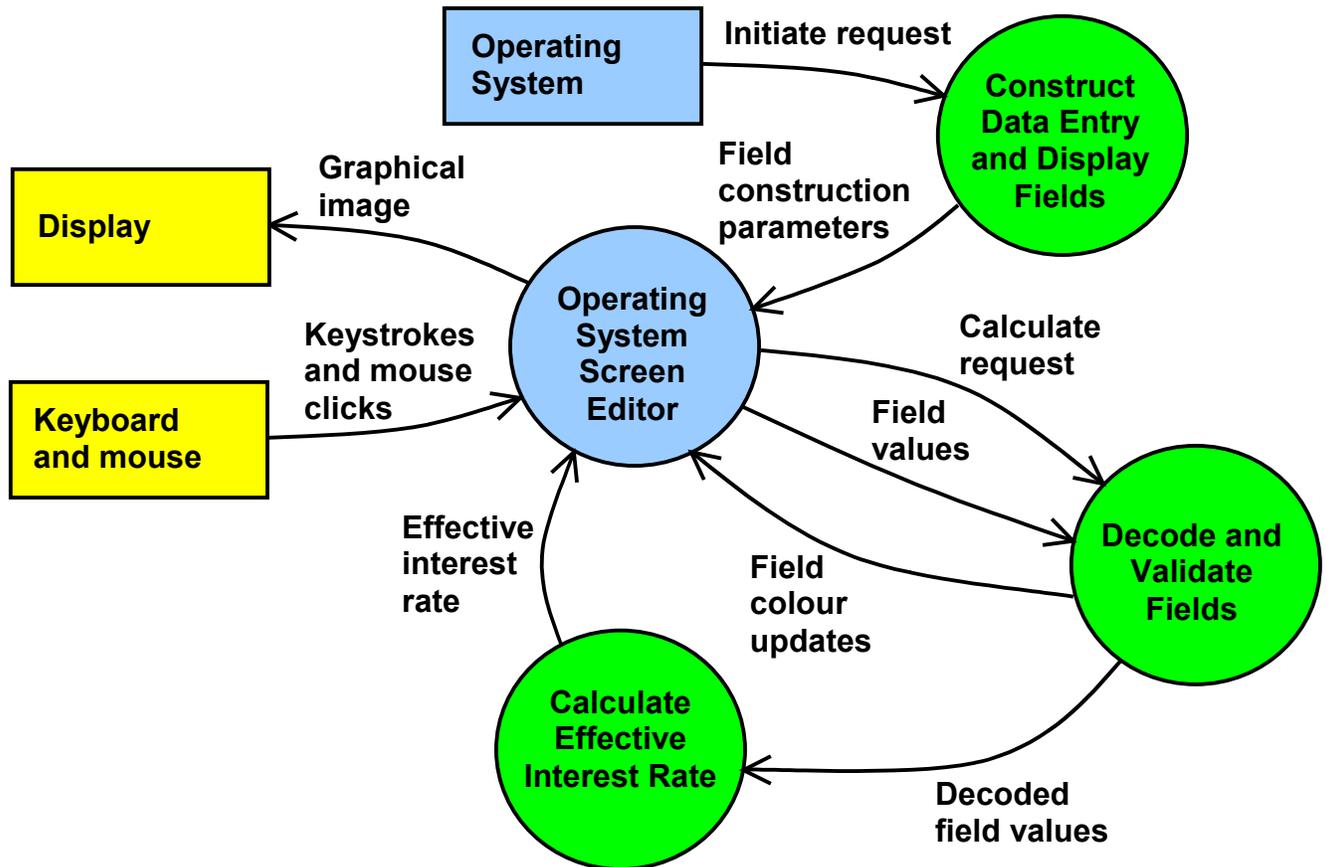| Language	Rating
Java	Good
A visual programming language (Visual Basic, Visual C++, Delphi, etc)	Good
C++ with MFC or equivalent GUI toolkit	Good
ANSI C	Bad
ANSI C++	Bad

The first critical decision we must make in programming the Loan Evaluator is to choose a programming language that provides data entry screen support.

Java and the visual programming languages would be good choices because they provide intrinsic support for setting up, reading data from and writing data to data entry screens.

C++ with the Microsoft Foundation Classes or an equivalent GUI toolkit would also have been an acceptable choice, and for much the same reason.

ANSI C and C++ would have been bad choices because neither language provides intrinsic support for data entry screens.

2.3 Dataflow Diagram

Provided we chose a programming language with intrinsic support for data entry screens, the data flows of the application would look something like these.

In the centre of the diagram is the operating system's screen editor. If we made a good choice of programming language, we won't have to program the screen editor, it will be provided by the operating system or language run-time system.

When the operating system initiates the program, it will call a procedure to construct the data entry and display fields. If we chose a visual programming language, the operating system will construct the fields from a table created by the screen definition tool. If we chose a language such as Java or C++ with the MFC, our program must make system calls to set up the data entry and display fields.

Once the data entry and display fields have been created, the operating system's screen editor can take responsibility for processing individual keystrokes and mouse clicks and for generating the graphical image that is shown on the terminal monitor.

When the user clicks the 'Calculate' button, the operating system's screen editor will generally send a calculate request back to the application.

When the application receives the calculate request, usually by the operating system calling a method in the application, the application must read and validate the values entered into the data entry screen. If any field values are found to be invalid, it should display those fields in red and return control to the operating system.

If all the entered field values are correct, the decoded field values should be passed to a process that calculates the effective interest rate and passes it back to the operating system's screen editor so that it can be displayed on the data entry screen.

2.4 Constructing the Data Entry Screen

```
    startField = new TextField [ MAX_PAY_TYPES ];
    periodField = new TextField [ MAX_PAY_TYPES ];
    quantityField = new TextField [ MAX_PAY_TYPES ];
    amountField = new TextField [ MAX_PAY_TYPES ];
    for (i = 0; i < MAX_PAY_TYPES; i++) {
        startField[i] = new TextField ();
        startField[i].setBounds (colWidth*20, y,
            colWidth*4, lineHeight*3/2);
        add (startField[i]);
        periodField[i] = new TextField ();
        periodField[i].setBounds (colWidth*30, y,
            colWidth*4, lineHeight*3/2);
        add (periodField[i]);
        quantityField[i] = new TextField ();
        quantityField[i].setBounds (colWidth*40, y,
            colWidth*4, lineHeight*3/2);
        add (quantityField[i]);
        amountField[i] = new TextField ();
        amountField[i].setBounds (colWidth*50, y,
            colWidth*12, lineHeight*3/2);
        add (amountField[i]);
        y += lineHeight * 2;
    }
```

This is the code that generates the table of data entry fields in the worked example.

There is no need to generate repeating sequences of fields one-at-a-time. The statements that create fields are not usually static declaration statements but executable system calls. They can be called in a loop just as well as they can be called in an in-line sequence.

In Java, the fields had to be created, in this case by instantiating a new TextField instance. The physical position and size of the field had to be set by calling the setBounds method, and then the field had to be embedded in the encompassing applet by calling the applet's add method. This sequence is repeated for each field in each row of the table, and for the amount borrowed and effective interest rate fields as well.

9

The system calls for setting up data entry screens are very language and operating system dependent, and the bookshops are full of paperbacks that describe with copious examples the ways in which the functions should be used. Learning how to set up data entry screens is something you can very easily pick up from a textbook.

2.5 Decoding and Validating the Fields

```
    // Decode the principal amount

    try {
        principal = Double.parseDouble
            (principalField.getText());
    } catch (NumberFormatException e) {
        principalField.setBackground (Color.RED);
        decodeFault = true;
    }
```

This code segment is typical of the decoding and validating that is required for each input field.

The parseDouble method converts the text string keyed into the data entry screen into a double precision floating-point number. If the text string did not represent a valid number, for example, if it contained a non-numeric character, parseDouble throws a NumberFormatException, which should be caught. On catching the NumberFormatException, the program can colour the erroneous field red as required by the specification and set a fault flag before going on to validate the other fields.

When all the fields have been validated, the fault flag can be tested to determine whether or not an attempt can be made to calculate the effective interest rate.

2.6 Calculating the Effective Interest Rate

$$P = \sum_{i=1}^{n} \frac{v_i}{\left(1 + \dfrac{R}{1200}\right)^{t_i}}$$

Where:

P	is the amount borrowed (the principal);
n	is the number of individual payments;
v_i	is the amount of an individual payment;
t_i	is the month number (i.e. time) at which the payment is to be made;
R	is the effective, monthly compound interest rate expressed as a percentage.

The only thing you couldn't pull out of a textbook was the solution to inverting the formula given in the question paper.

The formula was the classic formula for the present value of a series of payments. v_i is the value of the i-th repayment the borrower has to make to the bank. It is discounted by an interest rate R, which is changed from a percentage to a factor by a division by 100 and from an annual rate to a monthly rate by a division by 12 and converted into a multiplier by the addition of one. It is then raised to the power of t_i, which is the time, in months relative to the start of the transaction, when the payment has to be made. In effect, payments made in the future have a smaller cost than payments made now because you have the benefit of having the use of the money between now and then. Add up the present values of each of the repayments and we get the present value of all the repayments together.

However, the question did not ask us to calculate the present value of all the repayments given an interest rate. It asked us to find the interest rate that made the present value of all the repayments equal the amount borrowed. This is the effective interest rate of the loan, a measure by which the cost of alternative loans can be compared.

It is not hard to invert the present value formula when there is only a single payment (i.e. when 'n' is one). If 'n' is one, we can take logs of both sides of the equation and with a little algebraic manipulation can find R as a function of P.

But if 'n' has any value other than one, the summation frustrates the logarithmic transformation and, as far as I know, there is no easy algebraic solution.

Nevertheless, there are a number of computational methods that can solve the problem.

2.7 Sequential Search

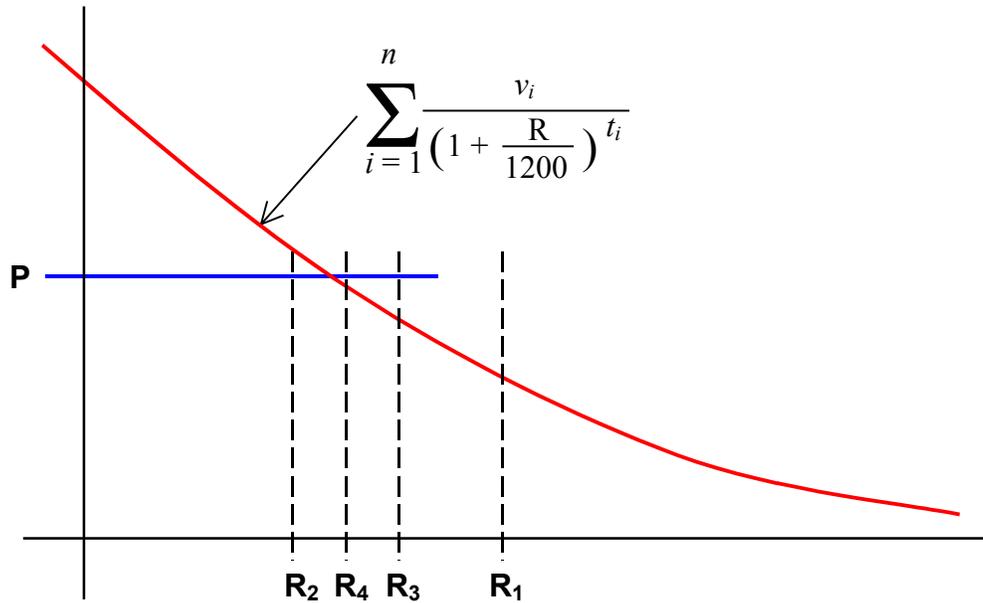$$\sum_{i=1}^{n} \frac{v_i}{\left(1 + \dfrac{R}{1200}\right)^{t_i}}$$

P

R
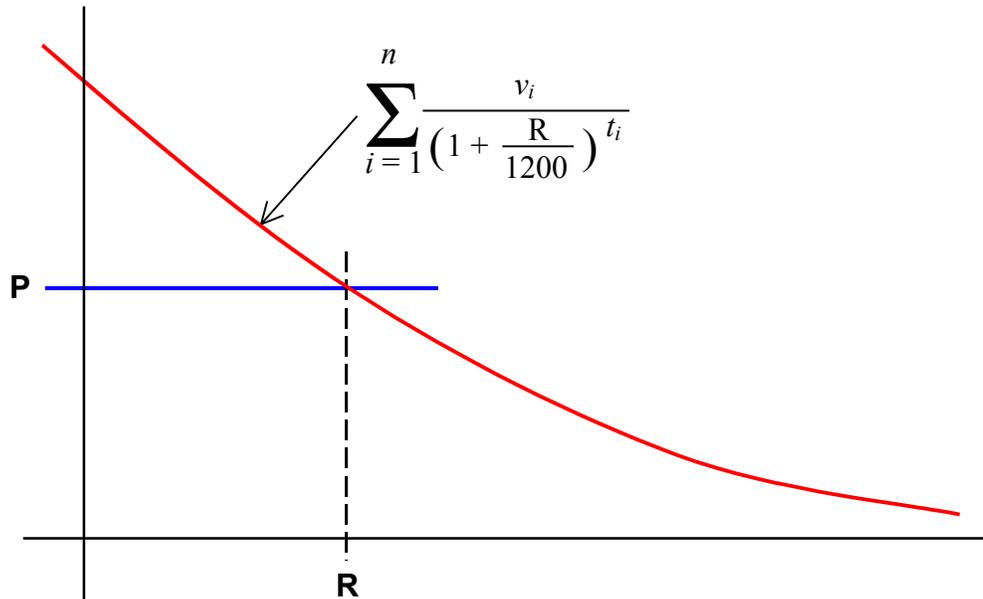
The simplest computational method is a simple sequential search.

The program can start with R equal to zero and test whether the sum is greater than the principal. If the sum is less than the principal, P, the specification allows the program to show asterisks in the effective interest rate field.

If the sum is greater than the principal, the program can test progressively greater values of R until the sum becomes smaller than the principal or R exceeds 100%. If R exceeds 100%, the program can show asterisks.

If a value of R is found which causes the sum to be smaller than the principal, the program will have two values of R, the value immediately before the sum went smaller, and the value immediately after the sum went smaller. The program could either choose the value that had the smallest difference between the sum and the principal, or linearly interpolate between the two values.

2.8 Binary Search

$$\sum_{i=1}^{n} \frac{v_i}{\left(1 + \dfrac{R}{1200}\right)^{t_i}}$$

P

R₂ R₄ R₃   R₁

(The figure contains the axes with the red curve, a horizontal blue line labeled P, the summation formula with an arrow pointing to a point on the curve, and labels R_2, R_4, R_3, R_1 along the horizontal axis.)

The algorithm used in the example solution is a binary search. It is significantly more efficient and accurate that the sequential search.

First the algorithm must check that the value of R lies in the range 0 to 100%, which can be done in the same way as in the sequential search. If the sum is less than the principal for an interest rate of 0 or the sum is greater than the principal for an interest rate of 100%, then the interest rate is outside the working range of the program and the program is permitted to show asterisks.

The binary search starts with a range from 0 to 100% and then progressively divides the range by cutting the range in half and determining the half in which the result lies.

For example, the program would start by testing R_1, which would be 50%. Because the sum is less than the principal at R_1, the program knows that R must lie between 0 and 50%.

The program would then test R_2, which is 25%. Because the sum is greater than the principal at R_2, the program knows that the answer must lie between 25% and 50%.

The program would then test R_3, which is 37.5%. Because the sum is less than the principal at R_3, the program knows the answer must lie between 25% and 37.5% and so on.

Each cycle of the binary search generates one bit of precision. We were asked to display the answer to two decimal places. Because the answer should be in the range 0 to 100%,

there will be two significant digits to the left of the decimal place and two significant digits to the right of the decimal place. That means we need a total of four significant digits of precision. There are slightly less than 4 bits per decimal digit, so 4 times 4 or 16 iterations of the binary search should give sufficient precision.

2.9 Newton's Method

$$\sum_{i=1}^{n} \frac{v_i}{\left(1 + \frac{R}{1200}\right)^{t_i}}$$

P

Tanget at R_2

Tanget at R_1

R_2 R_3 R_1
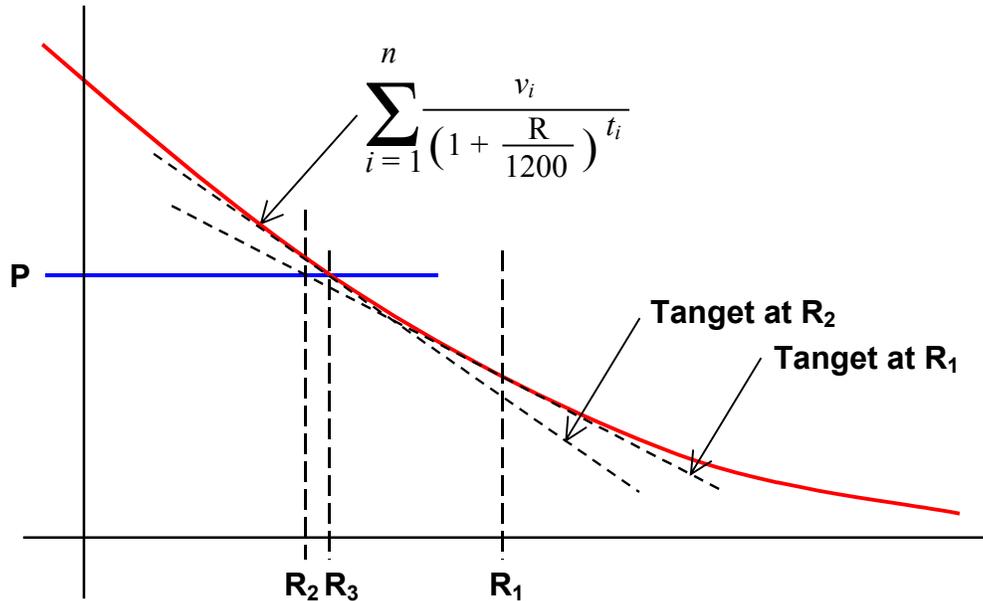
A third technique for inverting the formula is to apply Newton's method, named after Sir Isaac Newton who supposedly invented it.

To apply Newton's method, we start with a guess as to the value of R, say R_1 on the slide. We then find the point at which a line tangential to the sum at R1 passes through the horizontal line at height P. This gives us R_2, a better estimate of R. The process can then be repeated as many times as is needed to obtain the required precision.

The tangent can be found by calculating the value of the sum at R_1, which gives us a point through which the line passes, and the first differential of the sum at R_1, which gives us the slope of the line.

Newton's method will work for effective interest rates outside the range 0 to 100%, and should converge significantly faster than the sequential or binary search, but your calculus needs to be good enough to differentiate the sum and your algebra needs to be good enough to figure out the equation for the tangent and the point at which the tangent intersects the principal line.

2.10 Summary

The Loan Evaluator was the easiest problem I could think of that didn't have an obvious answer in a textbook; nevertheless it exercised some fundamental computer programming skills.

- Defining a data entry screen;

- Extracting data entered into the fields of a data entry screen;

- Decoding and validating data extracted from the fields of a data entry screen;

- Searching for the solution to an equation.

Would anyone like to ask any questions about the Loan Evaluator problem?

3 MICROCONTROLLER EMULATOR

3.1 Introduction
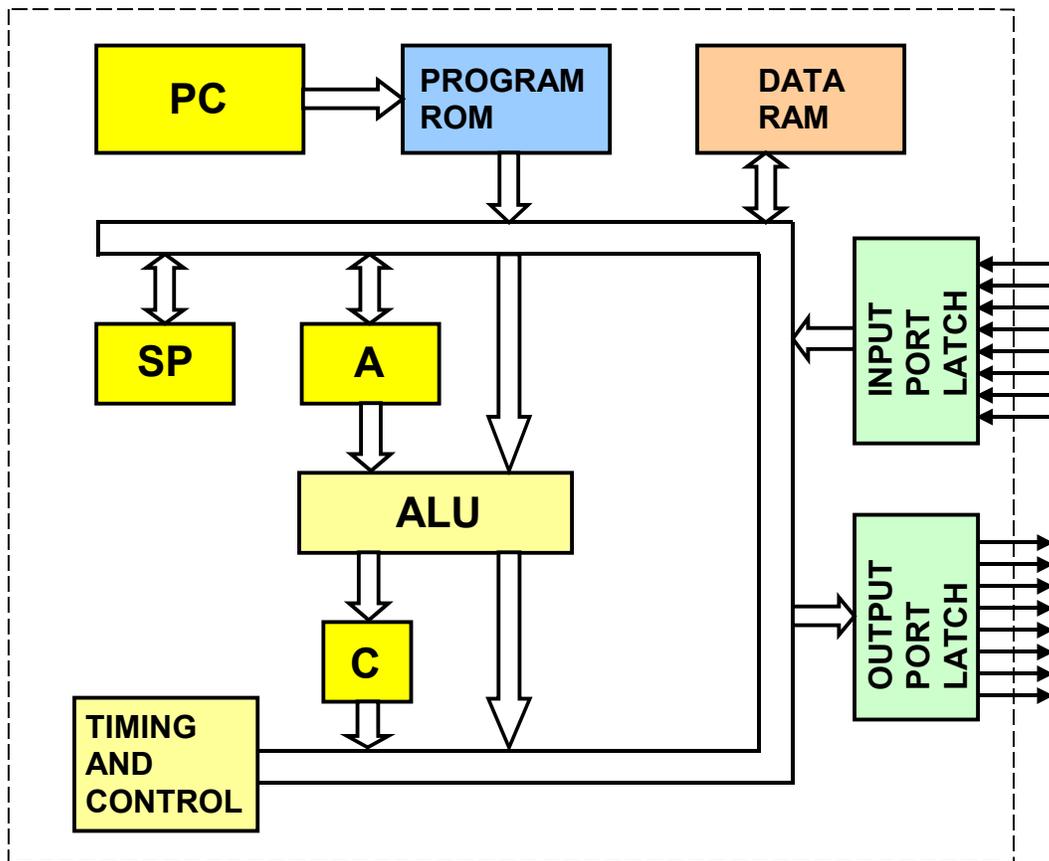
Now, let's look at slightly more difficult problem, the Microprocessor Emulator.

A microprocessor is an electronic circuit. This first slide is an imaginary block diagram of the microprocessor described in the question paper.

There is an internal 8-bit data bus, the program counter and the program ROM, the 240 bytes of data RAM, the accumulator, arithmetic and logic unit and carry bit, the stack pointer and the input and output ports.

When looking at the question paper, a good proportion of the verbiage is devoted to describing the instruction set. This might lead a naïve programmer to believe that all that has to be done is to program a virtual machine to process the instruction set, but that would be a mistake. Programming the virtual machine is quite straightforward and involves little more work than converting the instruction table into a switch statement.

The more difficult part of the question is dealing with the eight input and eight output signals.

3.2 Control Panel
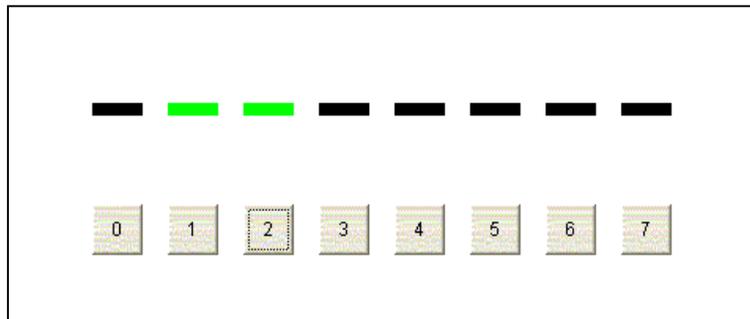
The question paper requires the emulator to have a control panel.

The indicators at the top of the control panel must be green when an output signal is one and black when the output signal is zero.

When one of the buttons at the bottom of the control panel is being pressed, the microprocessor must read one from the corresponding input port and when the button is released, the microprocessor must read zero from the input port.

Note that the input and output functions are independent. Microprocessor store operations determine the state of the indicators and the state of the buttons determine the result of a microprocessor load operation.

3.3 Dataflow Diagram
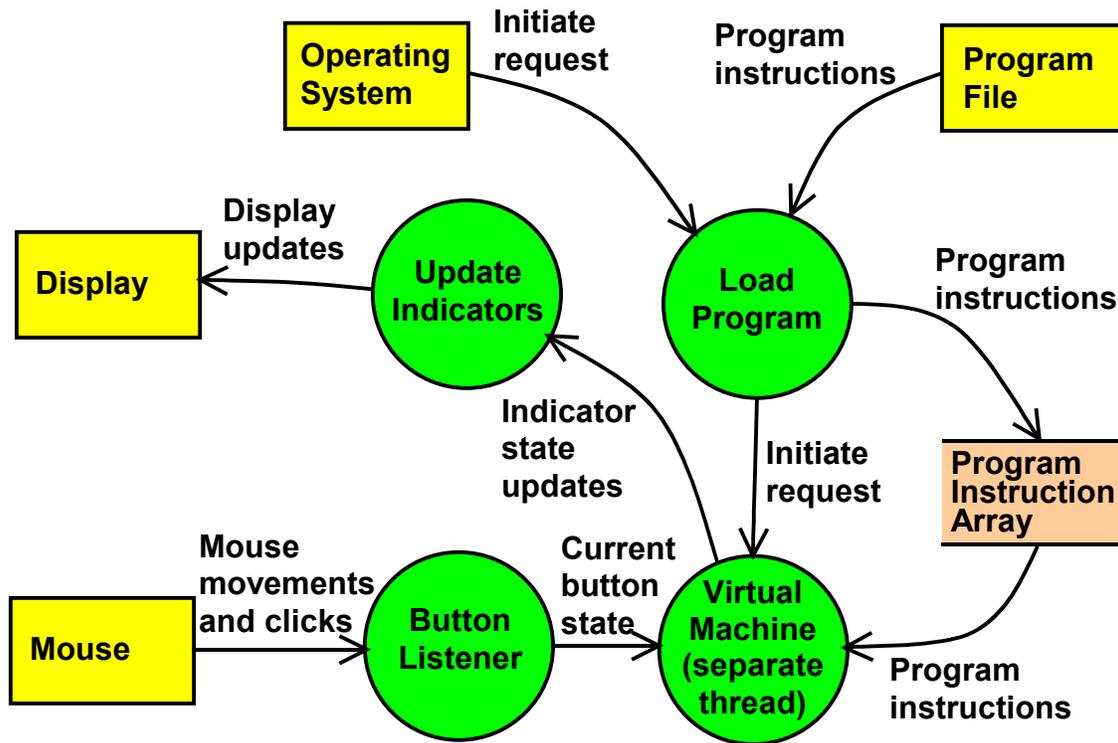
Before we can program the virtual machine, we need to consider how the virtual machine is going to communicate with the mouse and the display.

The main problem is that most operating systems of the kind that would be suitable for displaying the control panel assume that the application is event-driven. The operating system calls a method in the application to initiate the application and then waits, blocking all further stimuli to the application until the application returns from the initialisation method. Only when the application returns from the initialisation method will the operating system then call another method to indicate that a button has changed state.

If the application traps control in the initialisation method by running the virtual machine and not returning control back to the operating system, the operating system never gets an opportunity to report button state changes to the application. In effect, the virtual machine blocks its own input.

A good solution to this problem is to run the virtual machine in a separate thread as is shown in this dataflow diagram.

Before the virtual machine can start running, the program must be loaded from an external program file. This is simple enough; the only sensible way to store the program inside the emulator is in a byte array. The program loader should simply copy the program file into the byte array.

Once the virtual machine is started as a separate thread, our problems are still not entirely over. We need to devise a way to communicate the indicator state updates to operating

system's interface for painting the display panel and we need to devise a way to communicate the operating system's mouse activity stimuli to the virtual machine.

3.4 Update Indicators
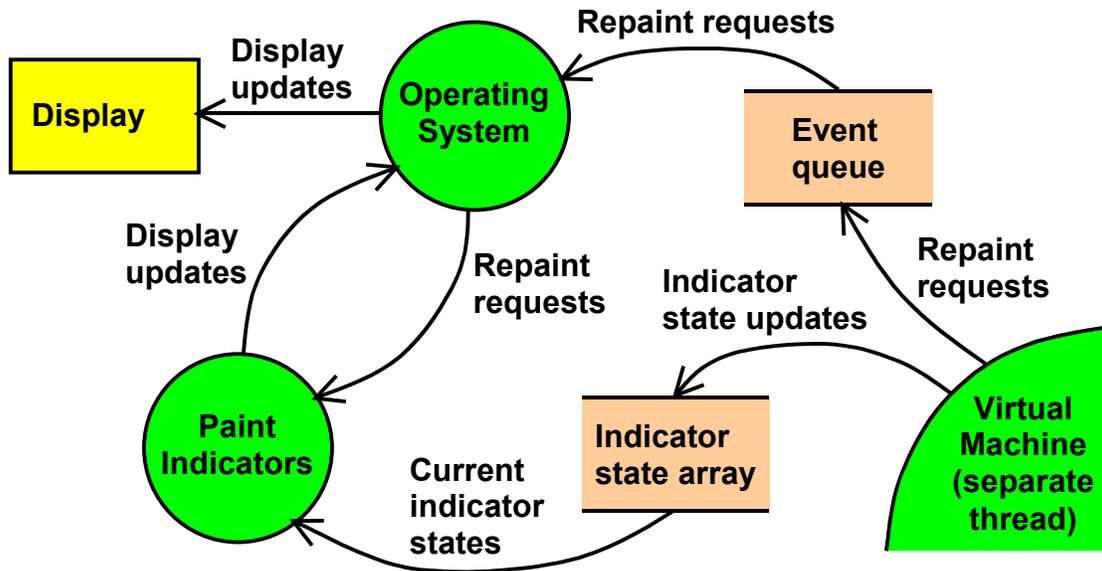
Let us now consider the problem of updating the indicators. Both Windows and Java call a paint method when they need an application to repaint a display. The paint method might be triggered by an internal repaint request from the application or an external trigger such as the window being made visible after it had been hidden.

In the sample solution, the virtual machine communicated updated indicator states by loading the new indicator state into an array and then sending a repaint request to the operating system to make the operating system repaint the display. The operating system would, in turn, call the application-level method to repaint the indicators.

The sample solution sent the repaint requests via the operating system's event queue to avoid any synchronisation problems that might arise from calling the operating system's repaint method from a free-running thread. I'm not sure that this was strictly necessary, though in my view it was good conservative programming.

3.5 Button Listener

Mouse movements and clicks

Mouse → **Button Listener** → **Button state array** → **Current button state** → **Virtual Machine (separate thread)**

Button state updates
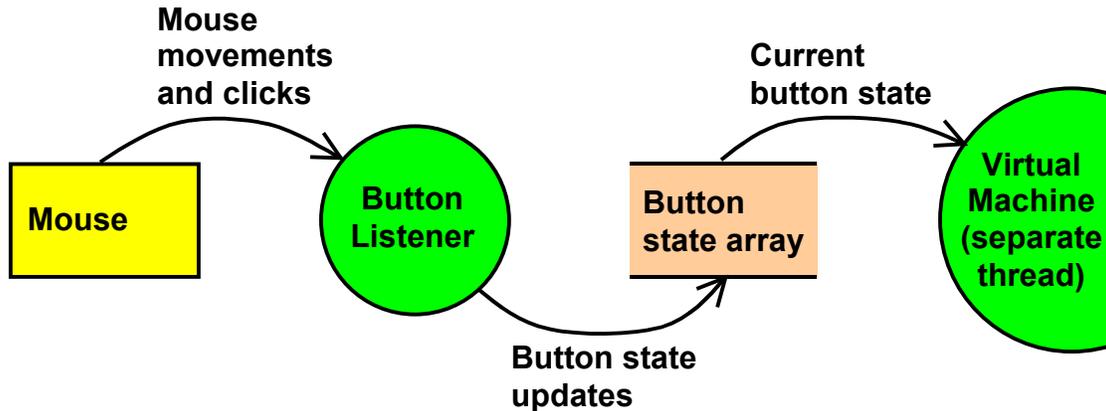
Similarly we need to buffer the current state of the buttons (pressed or released) with an array.

When the mouse is moved over a button and the mouse button is pressed or released, the operating system calls a method in the application, usually in an event-processing thread. The event-processing thread needs to communicate the current state of the button to the virtual machine thread. This can be accomplished by saving the current button state in an array that can be written by the button listener and read by the virtual machine.

Because the states of the buttons are reflected in the button state array and the states of the indicators are read from an indicator state array, the virtual machine can run freely, reading button states from the button state array and writing indicator states to the indicator state array as and when it needs to.

3.6 Register Set

C   **A**

SP

PC
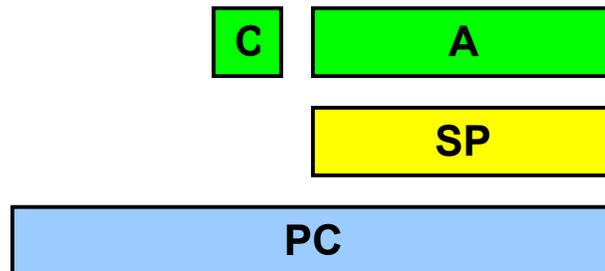
Now let's move on to considering the virtual machine.

The question paper explains that the microprocessor has three registers and a carry bit as shown on this slide.

The question paper says that the accumulator has 8 bits. Because the microprocessor has an 8-bit data address space, we can assume that the stack pointer will also have 8 bits and

because it has a 16-bit program address space we can safely assume that the program counter has 16 bits.

Nevertheless, we may not be able to use an 8-bit variable to emulate the stack pointer and a 16-bit variable to emulate the program counter.

If we're programming in Java, the 8-bit byte type is an 8-bit signed integer. An 8-bit signed integer stores values in the range –128 to +127. The stack pointer needs to store values in the range 0 to 255, being the addresses of the data address space. Therefore a byte data type is inappropriate for the stack pointer. I would be pragmatic and use an int variable to emulate the stack pointer.

Similarly, the 16-bit short Java data type stores values in the range –32768 to +32767. The program counter needs to operate in the range 0 to 65535, so again it would be better to use a 32-bit int variable to emulate the program counter.

3.7 Virtual Machine

```
byte       progMem[];      // Program memory
                           // (loaded by Program Loader)
boolean   c = false;      // Carry bit
int       a = 0;          // Accumulator
int       sp = 0;         // Stack pointer
int       pc = 0;         // Program counter

while ( ! Thread.interrupted()) {
     switch (progMem[pc]) {
     case 0x10:
          a = progMem[pc+1] & 0xff;
          pc += 2;
          break;
     // Process other instructions
     // ...
     }
}
```

With the exception of various complications arising from the use of signed variables, programming the virtual machine is quite straightforward. It's simply a loop and a switch statement.

In the example solution, the virtual machine thread was terminated by an interrupt sent from the operating system thread. The virtual machine polled the interrupt flag before executing each instruction to test when it needed to exit.

The switch statement switches on operation code. This slide shows the programming of the load accumulator, immediate instruction.

Because the program memory is stored in a signed byte array, it must be truncated to 8 bits by a logical-and operation before it is loaded into the accumulator. This is typical of the conditioning that needs to be applied to the processing shown in the instruction table.

3.8 Call and Return

Before	Instruction	After

CALL hi-lo

PC_0: data[--SP] = PC+3 & 255;
        data[--SP] = PC+3 >> 8;
        PC = hi << 8 | lo;
PC_0+3:

$lo(PC_0+3)$
$hi(PC_0+3)$

RET

PC = data[SP] << 8
        | data[SP+1];
SP += 2;

i.e.: PC = $(PC_0+3 >> 8) << 8$
        | $(PC_0+3 \& 255)$
        = $PC_0 + 3$
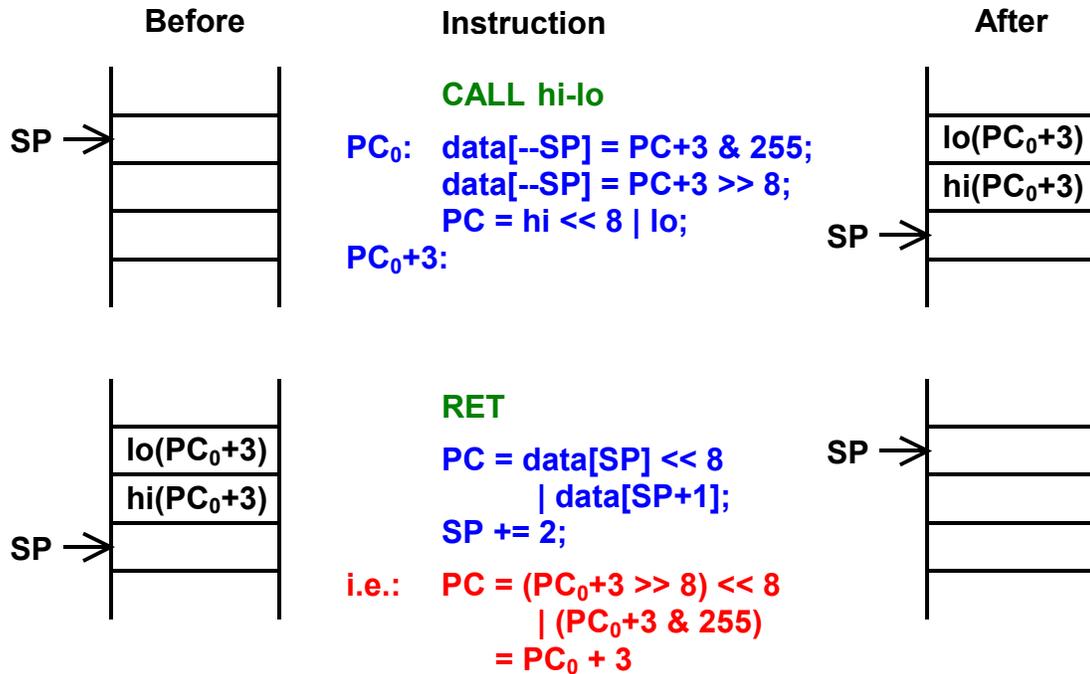
One of the main motivations for setting the Microprocessor Emulator problem was that it had become clear that very few graduates knew what a computer instruction set looked like, let alone the effect of more complex instructions such as call and return.

I would like to spend a little time explaining this, because it is something worth knowing and understanding.

The call instruction pushes the value of the program counter plus three onto the stack. Because the program counter is a 16-bit register, it needs two 8-bit data memory locations. It is pushed low-byte first and then high byte. The value of the program counter plus three is the address of the instruction after the call instruction. It is known as the 'return address'.

After pushing the return address on the stack, the microprocessor loads the program counter with the value 'hi-lo'. Hi and lo are the high and low bytes of the address of the subroutine respectively. Loading the program counter with the address of the subroutine causes the microprocessor to pass control to the first instruction in the subroutine.

When the subroutine executes a return instruction, it pops the high byte of the return address into the high byte of the program counter and the low byte of the return address

into the low byte of the program counter. Effectively loading the return address into the program counter and causing the program to resume execution at the instruction immediately after the original call instruction.

3.9 Summary

The Microprocessor Emulator problem was intended to be an exercise in multi-threaded programming and microprocessor instruction sets. It exercises these skills:

- Identifying processes by dataflow analysis.
- Displaying graphical objects.
- Receiving and processing button event messages.
- Loading the contents of a file into a byte array.
- Initiating and terminating a thread.
- Communicating between threads.
- Understanding a microprocessor instruction set.

Would anyone like to ask any questions about the Microprocessor Emulator problem?

4 TEAM SELECTION

Let us now discuss the most difficult question.

The background is that a government-linked company, Nautical Defence Systems or 'NDS' has been instructed to initiate a project to develop a fleet of state-of-the-art patrol boats.

NDS does not have the resources to develop the software systems for the patrol boats and is inviting outside contractors to bid for the work. NDS has defined a formula for rating each bidder that will presumably influence their selection of the final contractor.

We are working for one of those outside contractors and have been assigned the task of devising a computer program to optimise our company's rating by selecting the team from among the company's staff that maximises the value of the rating formula.

4.1 Team Rating Formula

$$Rt = \frac{\sum_{i=1}^{n} Rp_i}{Y^{(n-1)}}$$

Where:
- Rt is the rating of the team.
- n is the number of team members.
- Rp_i is the rating of the i-th programmer.
- Y is the constant 1.1

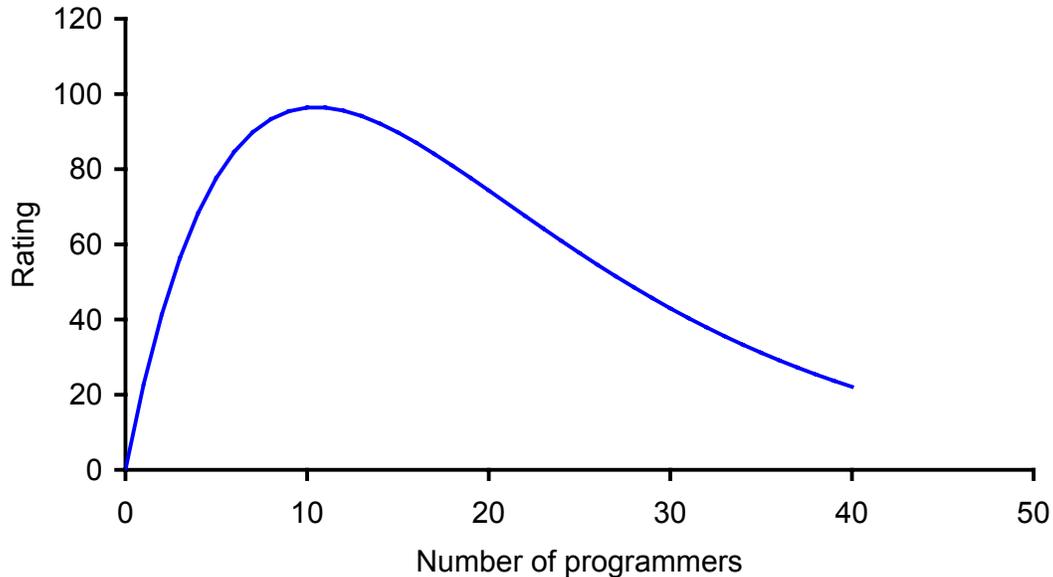The question paper provided this formula for rating the programming team.

Rp_i is the rating of an individual programmer.

n is the number of programmers.

Y is the constant 1.1.

4.2 Characteristics of the Rating Formula

Rating versus Team Size
Rp = 25 for all programmers

To contemplate the characteristics of the rating formula, we can plot the value of the rating formula versus the number of programmers where the rating of each programmer is constant, say in this case 25.

As we can see, the rating initially rises as the number of programmers increases, but then as the $Y^{(n-1)}$ denominator starts to have an effect, the rating starts to decline towards a zero asymptote.

4.3 Programmer Rating Formula

$$Rp_i = \sum_{j=1}^{m} V_{ij}$$

Where:
  Rp_i is the rating of the i-th programmer.
  V_{ij} is the value of the programmer's j-th project.
  m is the number of projects in the rating.

The question paper tells us that the rating of an individual programmer is the simple sum of the programmer's best five projects. If the programmer has less than five projects, the programmer's rating is the sum of all the projects the programmer undertook.

Consequently, we can write this formula for rating a programmer, where m is either 5 of the number of projects the programmer undertook and V_{ij} is the value of the individual projects.

I should think that it is reasonably obvious that we can optimise an individual programmer's rating by choosing to include his five highest-valued projects in the above formula.

4.4 Project Contention

If two or more programmers undertook a project jointly, the project may only be included in the rating of one of the programmers. The ratings of the other programmers must be derived from alternative projects.

However, life is never that easy in the E-Genting Programming Competition.

The question paper specified that if two programmers undertook a project jointly, only one of the two programmers could take credit for the project. The other programmer had to derive his rating from alternative projects.

I call this requirement 'project contention'. Our program is going to need a means to resolve project contention.

4.5 Database Schema

```
create table programmers (
    progNum         integer not null,
    progName        char(20) not null
);
create table projects (
    projNum         integer not null,
    projName        char(20) not null,
    projValue       double precision not null
);
create table assignments (
    asnProgNum      integer not null,
    asnProjNum      integer not null
);
```

The question paper provides us with the schema of a database that contains the information about the programmers and projects.
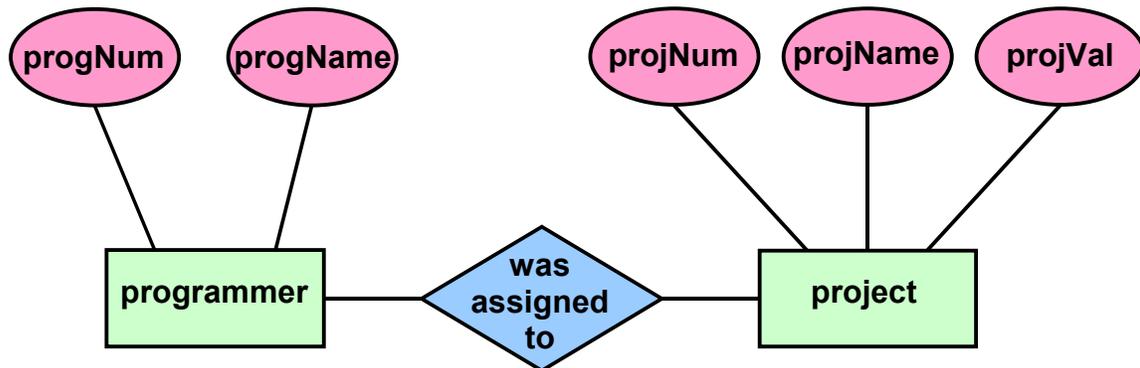
There are three tables.

The programmers table relates a programmer number to a programmer name.

The projects table relates a project number to a project name and a project value. The project value is the value that is needed to calculate a programmer's individual rating.

The assignments table associates programmers with projects.

4.6 Entity Relationship Diagram

A classical Chenian entity relationship diagram can help us understand how the database tables relate to each other.

There are two entities: programmers and projects, with attributes corresponding to the columns stored in the database tables.

Each row in the assignments table represents a relationship between a programmer and a project that indicates that the programmer was assigned to the project and conversely that the project was assigned to the programmer.

The Chenian entity relationship diagram makes it particularly clear that the contents of the assignments table are relationships as distinct to entities.

4.7 The Underlying Problems

- choose the projects to be included in each programmer rating;
- choose the programmers to be included in the team;
- resolve the project contention.

Solving the Team Selection problem reduces to solving three semi-independent problems.

We need to choose the projects to be included in each programmer rating so that each programmer receives the highest rating possible.

We need to choose the programmers to be included in the team and resolve the project contention so that the team receives the highest rating possible.

4.8 Munkres' Assignment Algorithm

The Munkres' Assignment Algorithm (sometimes referred to as the Hungarian Algorithm) assigns multiple jobs to multiple workers, each of whom may incur a different cost in completing each job, so as to minimise the total cost of completing all the jobs.

The Munkres' Assignment Algorithm, described by James Munkres in 1957, solves the problem of assigning 'n' jobs to 'm' workers, each of whom might incur a different cost for completing each task.

I won't go into describing the algorithm here, but if you are interested, there is a perfectly adequate entry for it in Wikipedia.

Munkres' Algorithm reduces the problem of finding the best assignments of jobs to workers from a problem that must be completed in exponential time to a problem that can be completed in polynomial time.

The problem is that Munkres' Algorithm assumes that both the number of jobs and the number of workers are constant. In the Team Selection problem, both the number of jobs and the number of workers are variables that are affected by the non-linear denominator of the team rating formula and its effect of choosing whether programmers are included in the team or not.

Certainly the non-linear denominator of the team rating formula frustrates a simple deployment of Munkres' Algorithm. Whether or not it frustrates use of the algorithm at all is a question that I would not attempt to answer in a one-day programming competition.

4.9 Brute Force

- *The database contains 75 programmers, 324 projects and 339 assignments.*
- At most 339 – 324 or 15 projects had more than one programmer.
- 2^{15} or 32,768 different contention alternatives need to be tested.
- 1 hour / 32,768 or around 100ms to test each alternative.

The question paper tells us that the database contains 75 programmers, 327 projects and 339 assignments.

This suggests that at most 339 – 324, or 15 projects were undertaken by more than one programmer.

Assuming that no, or very few projects were assigned to more than two programmers, 2^{15} or 32,768 different contention alternatives need to be tested.

The question paper tells us that the program must be designed in a manner that allows it to complete within one hour. One hour divided by 32,768 alternatives leaves us with approximately 100ms per alternative.

Provided our program is, in other respects, efficient, it seems quite likely that we can solve the project contention problem by simply testing every alternative, a method known colloquially as 'brute force'.

4.10 The Effect of Brute Force

Before applying brute force, 1 of:
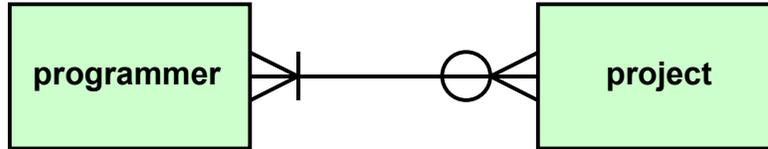
After applying brute force, 32,768 of:
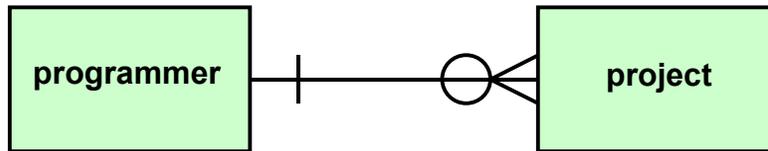
The affect of applying brute force to the contention problem is to translate the database from one instance of a database containing a many-to-many programmer-to-project relationship into around 2^{15} instances of a database containing a one-to-many programmer-to-project relationship, from which the best alternative can be found.

4.11 Getting the Best Programmer Ratings

For programmer 6:

projNum	projVal
100	5.6
052	4.7
128	3.9
054	3.8
027	3.2
107	1.7
079	1.2
132	1.1

$Rp_6 = 21.2$

In each database instance emitted by the brute force algorithm, each programmer has his own independent set of projects.

So to obtain the highest programmer rating for a programmer, all we have to do is to sort the projects assigned to the programmer by project value and select the five projects with the highest values.

4.12 Selecting the Best Team #1

$$Rt = \frac{\displaystyle\sum_{i=1}^{n} Rp_i}{Y^{(n-1)}}$$

Where:

Rt	is the rating of the team.
n	is the number of team members.
Rp_i	is the rating of the i-th programmer.
Y	is the constant 1.1

Now let us consider the problem of selecting the best programmers for inclusion in the team.

Looking at the team rating formula, for any given number of team members, 'n', the expression $Y^{(n-1)}$ will be constant. Consequently the best team will be the 'n' programmers with the best individual ratings.

This leads us to contemplate the idea that we might be able to sort the programmers in descending order of their individual ratings and then test different values of 'n' to find the best team.

4.13 Selecting the Best Team #2
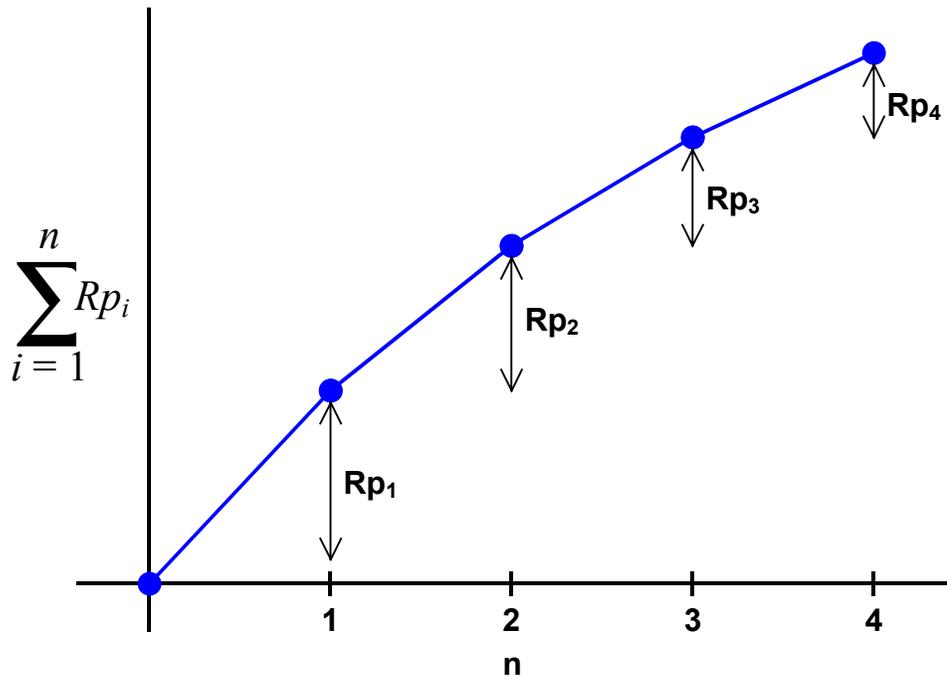
This slide is a plot of the sum of the individual ratings of the best 'n' programmers.

The important characteristic of this plot is that it always turns down. This is because the sort ensures that the rating of the i+1-th programmer is always less than or equal to the rating of the i-th programmer.

4.14 Selecting the Best Team #3

Rating Versus Team Size

The blue curve is labeled $\dfrac{n\,Rp_1}{Y^{(n-1)}}$

The red curve is labeled $\dfrac{\displaystyle\sum_{i=1}^{n} Rp_i}{Y^{(n-1)}}$

Because the plot of the sum always turns down, the plot of the team rating will always undercut the plot of n constant ratings by an ever increasing percentage as n increases.
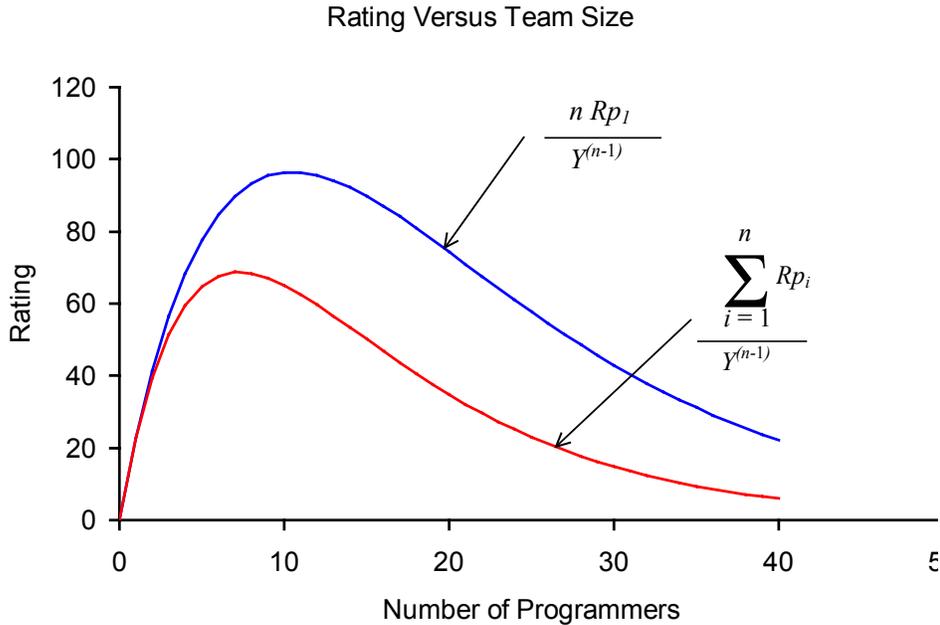
Because of this phenomenon and the fact that the curve for n constant ratings only has a single peak, which can be proved by taking the curve's first derivative and showing that the derivative is zero at one and only one value of n in the range zero to infinity, the plot of the team rating formula versus n will also always have a single peak.

This is important, because if the plot of the team rating versus n had more than a single peak it would be necessary to test all possible values of n to find the best value.

Because the plot of team rating versus n has only a single peak, all the Team Selection program has to do is to sort the individual programmer ratings in descending order and test the value of the team rating formula for progressively increasing values of n until the next value of the formula is less than the previous value of the formula. The previous value of n will then be the optimal value of n.

It seems quite likely that this can be completed in the 100ms per contention alternative

31

4.15 Dataflow Diagram
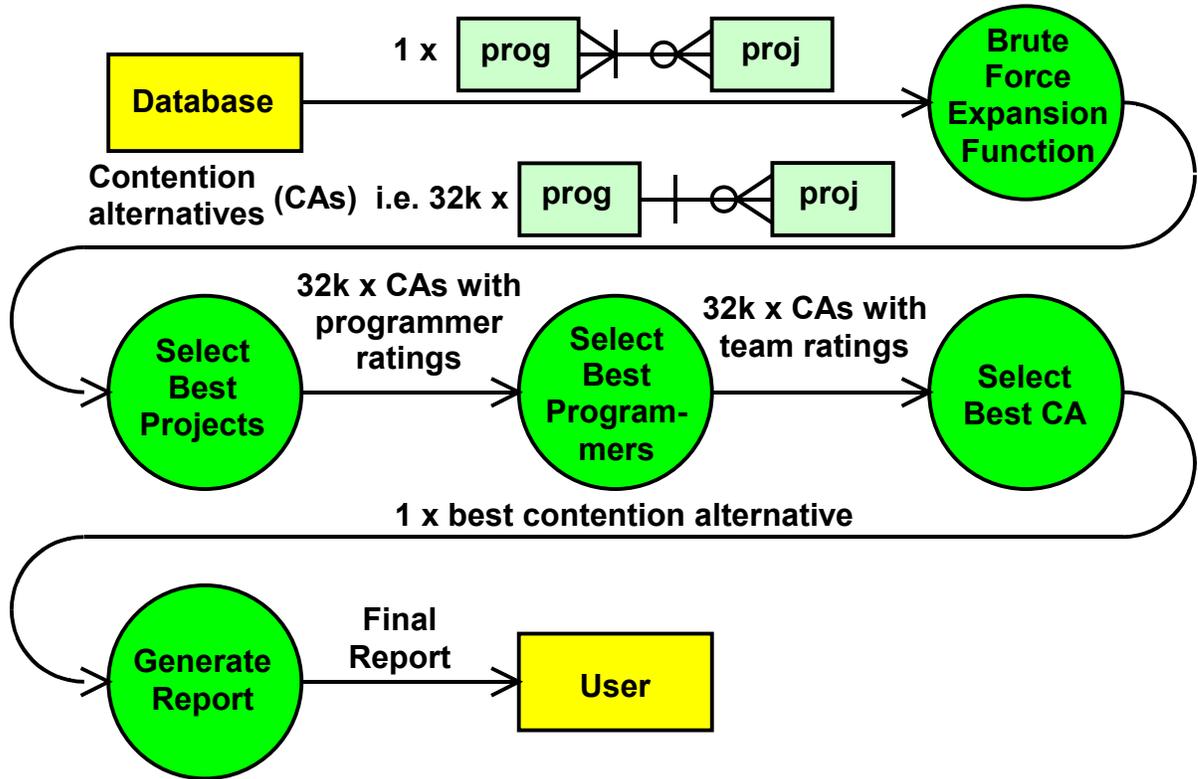
Putting it all together, our program has to do the processing shown on this slide.

It must extract the data from the database in the many-to-many form.

It must then put the data through a brute force expansion function that converts the single many-to-many data structure into around 32 thousand one-to-many data structures.

The one-to-many data structures must then pass through processes that select the best projects for each programmer, and select the best programmers for each contention alternative and then select the best contention alternative.

Finally a report generation process is needed to display the best contention alternative to the user.

4.16 Brute Force Expansion Function

```
Set of un-contended assignments;
Stack of contended projects;

Expand contended projects:
     If there are contended projects:
          Pop a contended project;
          For each programmer the project can be
          assigned to:
               Insert the programmer-project assignment;
               Recur;
               Delete the programmer-project assignment;
          End for;
          Push the contended project;
     Else
          Process the contention alternative;
     End if;
     Return.
```

We have already discussed how to select the best projects for each programmer, how to select the best programmers for each contention alternative and selecting the best contention alternative should be obvious. It's just the contention alternative with the highest team rating.

What we need to discuss is the means for converting the data structure stored on the database into the 32 thousand contention alternatives.

First, it is quite important to load the data stored in the database into a memory-resident data structure. Accessing a database tends to be a slow process. We do not want to be re-reading the database for each of the 32 thousand contention alternatives. That would consume our available time very quickly.

The algorithm for expanding the contended projects into the contention alternatives is essentially recursive.

Imagine the data structures are a set of un-contended assignments (this is the work-in-progress contention alternative) and a stack of contended projects.

The expansion function works by popping a contended project off the stack of contended projects and then looping through each alternative programmer to whom the project could be assigned.

In the body of the loop, the expansion function inserts the programmer-project assignment onto the set of un-contended assignments and then recurs to either expand the next contended project or process the contention alternative. When the recursion returns, the expansion function deletes the programmer-project assignment from the set of un-contended assignments to restore the set to its state before the recursion and then repeats the process for the next programmer who is contending for the project.

If there are no more contended projects, the expansion function falls through to process the contention alternative.

4.17 Summary

At this stage the only aspect of the Team Selection problem that we have not discussed is the generation of the report from the best contention alternative. I think, if you need help with that, you might be better off attempting a different question.

The Team Selection problem is a moderately difficult exercise in computer programming that required a reasonable amount of mathematical skill. A number of contestants attempted the question in the programming competition, but they all found provably sub-optimal ways of dealing with project contention. I am yet to find a better way than brute force.

The team selection problem exercises these skills:

- Database accessing using SQL.
- Using entity relationship diagrams to analyse and define data structures.
- Identifying core issues in a compound problem.
- Determining the feasibility of using brute force to solve investigative problems.
- Proving a solution is correct using mathematical theory.
- Using library sort functions.
- Using a linear search to find a peak value.
- Dataflow analysis and design.
- Expanding combinations using recursion.
- Report layout interpretation and report formatting.

Would anyone like to ask any questions about the Team Selection problem?

5 CASHIER ACTIVITY REPORT

5.1 Angina

According to Wikipedia:

> Angina pectoris is chest pain due to ischemia (a lack of blood and hence oxygen supply) of the heart muscle, generally due to obstruction or spasm of the coronary arteries (the heart's blood vessels). Coronary artery disease, the main cause of angina, is due to atherosclerosis of the cardiac arteries.
> *http://en.wikipedia.org/wiki/Angina*

Caused at least in part by the consumption of foods high in saturated fat and salt (e.g. hamburgers, french fries, fried chicken and pizzas).

Angina Marketing is, of course, selling heart disease, but that is someone else's problem. Our problem is to count their profits.

5.2 Background

- Angina Marketing operates several chains of fast food outlets. Their products include fried chicken, hamburgers and pizzas.

- Angina has a huge database table (over 200 million rows) that contains one row for each transaction processed at the fast food outlets.

- Angina needs a report to summarise the information in the transactions table.

- The transactions table has limited indexing.

- The reporting program should generate the report for a single day in less than 10 seconds.

- The program must do date validation and conversion.

- The opening and closing balance totals are not a simple total of the opening and closing balances of the individual shifts.

5.3 Database Schema

```
create table transactions (
     transNo              integer not null,
     tradingDate          char(10),
     locCode              char(6) not null,
     shiftNo              smallint not null,
     transType            smallint not null,
     transVal             integer not null
);
create index transNoInd on transactions(transNo);
```

The question paper gave us the schema of the transactions table.

The schema is defined in Structured Query Language known widely as SQL. SQL is by far the most widely used language for database definition, updating and accessing.

The two statements given in the question paper define a database table and a single index.

The database table contains a transaction number, which increments for each transaction.

The trading date is the date of the cashier shift. Each cashier shift is given a nominal trading day date. The trading day date is not necessarily the same as the real-time date. For example, a shift that starts at 10:00pm and finishes at 6:00am the next morning will typically have the trading day of the day when the shift starts, with the trading day rolling over at 6:00am the next day.

The location code identifies the cashier location. Each cashier location has a distinct location code.

The shift number identifies the shift within a trading day. In general shifts are given sequential numbers within each trading day. For example the 6:00am to 2:00pm shift might be shift 1, the 2:00pm to 10:00pm shift might be shift 2 and the 10:00pm to 6:00am shift might be shift 3. But we are told that shift numbers might be skipped. For example a cashier location might have data for shifts 1 and 3, but not shift 2.

The transaction type code identifies the type of transaction that is stored in the database row. I will discuss the transaction type codes further on the next slide.

The transaction value is the amount of the transaction in cents.

5.4 Transaction Type Codes

The question paper tells us that the transaction type codes are as shown on this slide:

Value	Name	Meaning
1	Opening balance	Money passed forward from the previous cashier shift.
2.	Fill	Money transferred to the cashier location to top up the float in the event that the cashier location does not have enough of a particular denomination of banknote.
3	Credit	Money transferred from the cashier location to the safe.
4	Sale	Money received from patrons in exchange for purchases.
5	Closing balance	Money passed forward to the next cashier shift.

5.5 Example Database Table

From the schema and the transaction type codes, we can imagine a transactions table that might look something like this:

transNo	tradingDate	locCode	shiftNo	transType	transVal
5179232	2007-01-01	FF-001	1	1 (open)	1060
5179233	2006-12-31	CP-005	3	1 (open)	1185
5179234	2007-01-01	FF-001	1	4 (sale)	275
5179235	2006-12-31	CP-005	3	4 (sale)	300
5179236	2007-01-01	FF-001	1	4 (sale)	270
5179237	2007-01-01	CP-010	2	1 (open)	960
5179238	2006-12-31	CP-005	3	4 (sale)	305
5179239	2007-01-01	FF-001	1	4 (sale)	300
5179240	2007-01-01	GB-007	1	1 (open)	955
5179241	2007-01-01	CP-010	2	4 (sale)	200
5179242	2007-01-01	FF-001	1	4 (sale)	265
5179243	2007-01-01	GB-009	1	1 (sale)	1905
5179244	2007-01-01	GB-007	1	4 (sale)	280
5179245	2007-01-01	GB-007	1	4 (sale)	295
5179246	2007-01-01	FF-001	1	4 (sale)	300
:	:	:	:	:	:

There is a sequential transaction number identifying each transaction.

The trading day date might switch between two dates depending on the time when the cashiers chose to roll the trading day forward.

The cashier location codes would be scattered depending on which cashier location processed the last transaction.

The shift numbers might also be scattered.

The transaction type codes indicate the type of transaction and the transaction value column indicates the value of the transaction.

5.6 Data Dictionary

1. date and time the report was printed;
2. reporting period (from-date and to-date);
3. for each cashier shift in the reporting period:
   a. cashier location code,
   b. trading day date,
   c. shift number,
   d. opening balance,
   e. total fills in the shift,
   f. total credits in the shift,
   g. total sales in the shift,
   h. closing balance;
4. totals for all shifts.

The question paper contained a data dictionary that specified the information that had to be displayed on the report.

This is a simplified version of the data dictionary.

The report had to show the date and time it was generated, and the reporting period. The reporting period was a from-date and to-date, which were the runtime parameters of the reporting program.

The data dictionary shows that the individual transactions stored on the transactions table had to be consolidated into a single row for each cashier shift. The report row has to show the cashier location code, trading day date, shift number, opening balance, total fills, total credits, total sales and the closing balance.

The data dictionary also shows that totals are required for all the selected shifts.

The question paper specified that the report had to be sorted by cashier location code, trading day date and shift number.

5.7 Balance Calculation

Location	Date	Shift	Opening balance	Fills	Credits	Sales	Closing balance
FF-001	24-09-06	1	1,200.00	100.00	2,500.00	2,700.00	1,500.00
FF-001	24-09-06	2	1,500.00	0.00	2,800.00	2,500.00	1,200.00
FF-002	24-09-06	1	800.00	200.00	1,700.00	1,950.00	1,250.00
FF-002	24-09-06	2	1,250.00	0.00	2,200.00	1,850.00	900.00
TOTAL			2,000.00 (1,200+800)	300.00	9,200.00	9,000.00	2,100.00 (1,200+900)

As a slight complication, the question paper advised that the opening and closing balance totals are not simple sums of the opening and closing balance columns. The total opening balance is the sum of the opening balances of the first shifts of each cashier location and the total closing balance is the sum of the closing balances of the last shifts of each cashier location.

As a further complication, if the last shift is still open (i.e. there is no closing balance transaction on the database) the reporting program must calculate the current balance at the location from the opening balance, fills, credits and sales and display it in the closing balance column.

5.8 Report Layout

The question paper specified how the report was to be laid out.

The report layout was similar to this:

```
                1         2         3         4         5         6
         1...5....0....5....0....5....0....5....0....5....0....5....0..
         DD-MM-YY HH:MM    CASHIER ACTIVITY REPORT           PAGE X-X
                           for XX-XX-XX to XX-XX-XX

         Cashier             Opening                            Closing
         Loc     Date    Shift balance  Fills  Credits   Sales  balance

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

The various fields in the data dictionary had to be printed at the locations indicated by Xs on the report layout.

The date and time had to be displayed in day-day, month-month, year-year, hour-hour, minute-minute format and a page number had to be displayed on the top right of each page.

The numbers across the top of the layout were not to be printed on the report. They were there to help the contestants figure out the position of each field.

The dollar values had to be shown to two decimal places and had to include a comma thousands separator.

5.9 Prototype Report

From the data dictionary and the report layout we can figure out that the report needs to look something like this:

```
07-12-06 11:05      CASHIER ACTIVITY REPORT            PAGE   1
                    for 24-09-06 to 24-09-06


Cashier                  Opening                          Closing
Loc     Date     Shift balance  Fills  Credits    Sales  balance


FF-001 24-09-06   1  1,200.00 100.00 2,500.00 2,700.00 1,500.00
FF-001 24-09-06   2  1,500.00   0.00 2,800.00 2,500.00 1,200.00
FF-002 24-09-06   1    800.00 200.00 1,700.00 1.950.00 1.250.00
FF-002 24-09-06   2  1,250.00   0.00 2,200.00 1,850.00   900.00
                     -------- ------ -------- -------- --------
TOTAL                2,000.00 300.00 9.200.00 9,000.00 2,100.00
                     ======== ====== ======== ======== ========
```

Each of the fields marked by Xs in the report layout is replaced with the value of an item in the data dictionary.

5.10 Analysing the Data
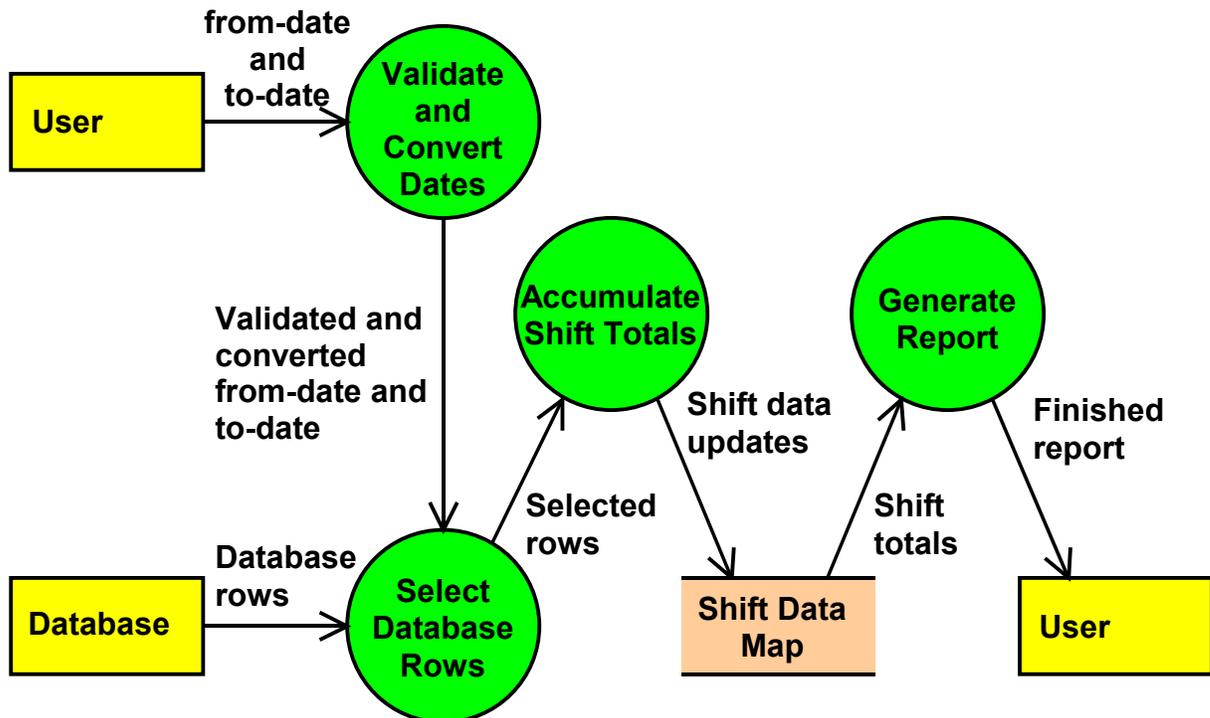
Having now come to an understanding of what the question entails, we are in a position to sketch a dataflow diagram that identifies the processes that we have to create.

The user provides a from-date and a to-date. These dates need to be validated and converted into a form in which they can be used to select database rows.

We need a process to select the database rows. This involves little more than selecting all the rows that lie within the range of the from-date and to-date, but the performance requirements make this a non-trivial task.

The selected rows then have to be accumulated into shift totals, which can be accumulated in an internal data structure. A map keyed by cashier location, trading day date and shift would be a good choice, as it would satisfy both the transaction accumulation and sorting requirements.

Then we have a report generation process that takes the data loaded into the shift data map and produces the report.

5.11 Date Conversion

```
struct Date_c {
        // Private declarations...
public:
        bool DateFromISO (const char *isoDate);
                // Load the date from an ISO-8601 format
                // date string (YYYY-MM-DD).  Return 0
                // if the string is invalid.
        bool DateFromUser (const UserDate_t userDate);
                // Load the date from a user format date
                // string (DD-MM-YY).  Return 0 if the
                // string is invalid.
        char *DateToISO (char *isoDate);
                // Convert the date to ISO format.
        char *DateToUser (UserDate_t userDate) const;
                // Convert the date to a user format.
        bool operator < (const Date_c &date) const;
                // Compare dates
        Date_c &operator -- (int);
                // Postfix decrement operator
        Date_c &operator ++ (int);
                // Postfix increment operator
};
```

C++ library support for dates is quite weak. So, if we are to solve this problem using C++ we are going to have to create a suite of date manipulation functions to perform the various date validation and conversion functions.

This is the declaration of the class in the sample solution.

The DateFromISO function converts an ISO YYYY-MM-DD format date into the internal format stored in the class instance data.

The DateFromUser function converts the user DD-MM-YY format date into the internal format.

The DateToISO and DateToUser functions convert the date stored in the class instance data into the ISO and user format respectively.

The less-than operator compares dates.

The decrement and increment operators provide some limited date arithmetic. The decrement date rolls the date back one day and the increment operator rolls the date forward one day.

Programming these functions is quite straightforward and I would hope that every student would be able to do it by the end of the first year.

5.12 Naïve Select Statement

A naïve implementation of the Cashier Activity Report might select the database rows by an SQL statement as simple as this:

```
select      tradingDate, locCode, shiftNo,
            transType, transVal
from        transactions
where       tradingDate is between
                :fromDate and :toDate;
```

But the question paper specified that the reporting program should generate the report for a single day in less than 10 seconds.

We were also told that 'any attempt to search on a key other than the transaction number results in a serial scan of the table, which takes around 2.5 hours'. The naïve select statement searches the database for specific trading day dates, which are not transaction numbers, so we should expect the naïve select statement to take 2.5 hours to execute.

That said, the performance requirement was a should-level requirement rather than a must-level requirement, so contestants who used the naïve select statement to select the database rows were marked down rather than disqualified.

5.13 Fuzzy Index

The question paper gave us a clue, that the transaction number index might be able to be used as a 'fuzzy index' for trading day date. It told us that 'at any point in time when a cashier was trading in day d, all the other cashier locations are trading in either the day before d, the day d, or the day after d.

42

transNo	tradingDate	
5179234	2007-01-01	
5179235	2006-12-31	
5179236	2007-01-01	Because this date is 2007-01-01,
5179237	2007-01-01	this date must be after 2006-12-31
5179238	2006-12-31	
5179239	2007-01-01	
5179240	2007-01-01	
5179241	2007-01-01	This date must be before 2007-01-02,
5179242	2007-01-01	because this date is 2007-01-01.
5179243	2007-01-01	
5179244	2007-01-01	
:	:	

What this means is that if we find a particular date on the database, the date that follows it when the rows are sorted in transaction number order cannot be earlier than one day before the date of the preceding row.

Similarly, if we find a date on the database, the date that precedes it when the rows are sorted in transaction number order cannot be later than one day after the date of the following row.

This means that if we can find a transaction with a date two days before the from-date, all the rows we need to extract will have transaction numbers greater than that of the transaction we found.

Similarly, if we find a transaction with a date two days after the to-date, all the rows we need to extract will have transaction numbers less than that of the transaction we found.

In fact, the only transaction we actually have to find is a transaction with a date two days before the from-date, because we can then scan forward through the transactions table in transaction number order until we retrieve a row containing a date that is two days after the to-date.

5.14 Binary Search

```
    lTransNo = first transaction no on database;
    hTransNo = last transaction no on database;
    startDate = fromDate minus one day;
    while (lTransNo < hTransNo) {
         mTransNo = (lTransNo + hTransNo) / 2;
         exec sql  select    tradingDate
                   into      :tradingDate
                   from      transactions
                   where     transNo = :mTransNo;
         if (SQLCODE != 0)
              /* error processing */;
         if (tradingDate < startDate)
              lTransNo = mTransNo + 1;
         else
              hTransNo = mTransNo;
    }
    // lTransNo contains the transaction number of the
    // first row that needs to be tested.
```

This is the algorithm used to discover the start of the sequential scan of the transactions table in the example solution. It is an adaptation of the binary search algorithm, the same algorithm we used to solve the Loan Evaluator problem. There you go, learn one algorithm and solve two problems. It's a bargain.

We can prove that lTransNo is never greater than the transaction number of the first row that needs to be processed, because the row before lTransNo has a trading day date two or more days before the from-date.

The binary search moves lTransNo close to the optimal threshold for starting the search, but never past it.

5.15 Scanning the Transactions Table

```
    exec sql declare transCur cursor for
        select    transNo, tradingDate, ...
        from      transactions
        where     transNo >= :lTransNo
        order by  transNo;
    exec sql open transCur;
    for (;;) {
        exec sql fetch transCur
                 into :transNo, :trDate, ...;
        if (SQLCODE != 0) break;
        if (trDate > toDate plus one day) break;
        if (trDate >= fromDate && trDate <= toDate)
            /* add the row to the shift totals */;
    }
    exec sql close transCur;
```

To extract the data from the database, we can do a sequential scan in transaction number order, starting from the threshold found in the binary search. The scan can function efficiently because it is in the same sequence as the transaction number index.

The scan can end at the end of the transactions table or when a transaction date is discovered that is two or more days after the to-date.

Out-of-scope rows within the scan range can be filtered out with a simple 'if' statement.

5.16 Accumulation of Shift Totals

```
      rowKey.rowDate = trDate;
      strcpy (rowKey.rowLocCode, locCode);
      rowKey.rowShiftNo = shiftNo;
      rowVal = &rowMap[rowKey];
      switch (transType) {
      case TT_OPENING_BALANCE:
          rowVal->rowOpenBal = transVal;
          break;
      case TT_FILL:
          rowVal->rowFills += transVal;
          break;
      case TT_CREDIT:
          rowVal->rowCredits += transVal;
          break;
      case TT_SALE:
          rowVal->rowSales += transVal;
          break;
      case TT_CLOSING_BALANCE:
          rowVal->rowCloseBal = transVal;
          rowVal->rowClosed = 1;
          break;
      }
```

Accumulating the shift totals is quite straightforward. If we create a map that is keyed by trading date, location code and shift number, the opening and closing balances can be assigned to the appropriate fields when they are located and fills, credits and sales can simply be totalled up.

The closed flag for the closing balance tells us whether it is necessary to calculate a theoretical closing balance for an open location from the opening balance, fills, credits and sales.

5.17 Calculating Totals

```
    prevLoc = null;
    prevCloseBal = 0;
    for each shift row in location code,
    trading date and shift number order:
        if necessary, calculate the closing balance;
        totalFills += row.fills;
        totalCredits += row.credits;
        totalSales += row.sales;
        if prevLoc is null or prevLoc != row.locCode:
            totalOpenBal += row.openBal;
            if prevLoc is not null:
                totalCloseBal += prevCloseBal;
            prevLoc = row.locCode;
        end if;
        prevCloseBal = row.closeBal;
    end for;
    if prevLoc is not null:
        totalCloseBal += prevCloseBal;
```

The one remaining trap is to correctly calculate the total opening and closing balances in the manner that we discussed earlier.

This pseudo-code explains how it was done in the example solution.

There is a main loop that processes the rows in the shift data map in location code, trading date and shift number order.

If necessary, the closing balance for the shift can be calculated from the opening balance, fills, credits and sales.

The fills, credits and sales, can be summed in the conventional way.

Two variables, a previous location code and a previous balance help the program correctly accumulate the opening and closing balances.

The central 'if' block is executed for the first iteration and each time the location code changes. This causes it to be executed for the first shift for each cashier location, which is the condition for accumulating the opening balance values.

The closing balance is the total of the closing balances of the last shifts for each cashier location. To calculate the closing balance, a previous closing balance variable carries the closing balance of the previous row forward to the next iteration. Whenever the location code changes, the previous closing balance is added to the closing balance total. At the end of the loop, the closing balance of the last shift for the last cashier location is added to the total closing balance.

5.18 Formatting the Report

Common formatting mistakes:

- not positioning each field at the right position on the page;

- displaying numeric fields without commas when commas are specified;

- not displaying a page title at the start of each page;

- not right justifying numeric columns and left justifying text columns;

- forgetting to display totals.

Generating the report should be a simple matter of formatting the contents of the map, but it is very rare for a contestant to do it well.

These are some of the common mistakes made by contestants.

All I can say is that we would rather not employ careless programmers. If you want to work for a good company, you are going to need to show that you are careful in your approach to the tasks you are given to complete.

5.19 Summary

The Cashier Activity Report is a middle-of-the-road commercial reporting program. None of the problems it presents are particularly difficult to overcome when looked at individually, but combined quite a few of these relatively easy-to-solve problems, and that tends to make it a little more difficult. The problem exercises these skills:

- Database accessing using SQL.

- Data dictionary interpretation.

- Report layout interpretation.

- Dataflow analysis.

- Date validation and conversion.

- Using a binary search to overcome DBMS performance limitations.

- Correctly totalling carry-forward and brought-forward financial statistics.

- Formatting a report to conform to a specification.

To paraphrase Mark Twain, they are skills worth your learning.

Would anyone like to ask any questions concerning the Cashier Activity Report?