THE UNIVERSITY OF WAIKATO

COMPUTER SCIENCE SCHOLARSHIP EXAMINATION

2005

PRACTICAL SECTION

TIME ALLOWED:	Six hours with a break for lunch at the discretion of the supervisor
NUMBER OF QUESTIONS IN PAPER:	Three
NUMBER OF QUESTIONS TO BE ANSWERED:	Three
GENERAL INSTRUCTIONS:	Candidates are to answer ALL THREE questions. All questions are important. Answer as much of each question as you can. Plan your time to allow a good attempt at each question, but be aware that Question 3 is the most difficult and will take considerably longer than the others.
SPECIAL INSTRUCTIONS:	Please hand in listings, notes and answers to written questions, and a floppy disk with your program/computer work for each question. Please make sure that copies of programs are stored as plain text files. You cannot assume that the examiner has available any special software that might be required to read your files. Candidates may use any texts or manuals for reference during the examination.
CALCULATORS PERMITTED:	Yes

1. **A Special What-If (Spreadsheet Use)**

In this question you are asked to use a spreadsheet to do calculations and to display the results. We expect that the spreadsheet will be used for all calculations - you will be marked down for performing calculations by hand and directly entering the results. Your work will be graded on three criteria.

(a) The accuracy of your results.

(b) The skill you show in making use of the capabilities of the spreadsheet.

(c) The presentation of your results. We have deliberately not provided any instructions concerning layout or formatting

Spreadsheets are often used for 'what-if' calculations. This question asks you to put yourself in the position of someone analysing the New Zealand election results on the day after the election. On that day we knew the results from all the ordinary votes, but 10% of voters had cast 'special votes', and these had yet to be counted. The final result would not be known for two more weeks. News reporters and party strategists were busy with questions like: "What if the Green Party did well in the special votes?" Your task is to build a spreadsheet to help answer some election what-if questions.

The results on the election night were as follows (excluding small parties). You also need to know that the number of special votes was 228958.

Party	Votes on election night	Electorate seats won
Labour	832425	31
National	809674	31
NZ First	119336	0
Green	103617	0
Māori	40488	4
United Future	55605	1
ACT NZ	31074	1
JAP	24624	1

Your spreadsheet should

1. Calculate the proportion of votes attained by each party. (eg: Labour 41.27%, etc)
2. Allow for a person using your spreadsheet to enter their guesses as to the proportion of special votes each party might receive. (eg: They might guess 30% for Labour, 70% for National and none for the others.)
3. Calculate an estimate of the final vote for each party as the number on-the-night plus their proportion of the specials.

   Note: This calculation may give you fractions of a vote – eg: 30% of the specials is 589687.4 votes. For the purposes of this calculation that doesn't matter.
4. Calculate how many seats in parliament would be awarded to each party. You may assume that the special votes will not affect the number of electorate seats won. The algorithm used for doing the seat calculation is detailed on the next page.
5. Display a chart (either a pie or a bar chart) showing seats in parliament for each party.

QUESTION 1 CONTINUED ON NEXT PAGE

Seat Allocation Algorithm

In the New Zealand parliament there are usually 120 seats. Some election results can require an increase in the number of seats (as happened this year), but the calculation always starts with the goal of allocating 120 seats.

1. Parties that both fail to gain an electorate seat and also fail to get 5% or more of the vote, get no seats.

The idea in a proportional representation electoral system like New Zealand's is that parties should be given the same proportion of seats in parliament as they receive votes in the election. There are 120 seats available. So, if Labour and National were the only parties and each received exactly 50% of the vote, each would be given 60 seats (50% of 120). In practice things usually don't work out so neatly. What would happen if Labour got 51% of the votes and National 49%? The simple calculation gives Labour 61.2 seats and National 58.8 seats. But it isn't possible to have 0.2 of a member of parliament. Each party must be given a whole number of seats. We require an algorithm that gives whole numbers of seats, keeping the proportions as accurate as possible. New Zealand elections use a method called the Sainte-Laguē system.

The calculation process we describe here is not the same as that actually used, but is better suited to a spreadsheet. We will first outline the algorithm and then suggest a way of doing it in your spreadsheet.

2. Algorithm: Let R be a variable. Start with R = 120. Calculate a number of seats for each party by working out their shares of R seats (which will usually involve a fraction) and rounding them off to the nearest integer. (When rounding, fractions less than 0.5 go to zero, and those greater than or equal to 0.5 go to one.) Total the number of seats allocated. If the result is 120 we have finished. Usually the result will be too low or too high. For example, if three parties received 61%, 17% and 22% of the vote respectively, the calculation would lead to them getting 73.2 => 73, 20.4 => 20 and 26.4 => 26 seats respectively, for a total of 119 – one too few. The idea is that we raise or lower R until we get a satisfactory result. In the example, if we set R to 121, then we overshoot and allocate 122 seats. The value of 120.47 works nicely.

   Spreadsheet: Experimentation with values is hard to program in a spreadsheet, but can easily be done manually. Just provide a cell for R. Allow the person using the program to adjust the value until they get an allocation of seats that works properly.

3. The final number of seats allocated to each party is either the number chosen in step 2 above, or the number of electorate seats gained by that party – whichever value is the larger. This step can (and does) lead to parliament having more than 120 seats.

Hint: The Excel function 'Int' will remove the fractional part of a number, eg: Int(27.8) = 27. You can use this as a basis for rounding. Other spreadsheets will have similar functions.

CONTINUED

2. **Long Distance Badminton (Careful and Accurate Programming)**

Your programming work in this question will be assessed on two criteria:

(a) Completeness and accuracy of the program.

(b) Good presentation. That is, it should make good use of programming language facilities, be well organised, neatly laid out, and lightly commented.

Data communication between the towns of Lonelyville and Fardistanton is very slow. In a few weeks the Lonelyville badminton team will be going to Fardistanton for the annual tournament. The people of Lonelyville want to receive progress scores during each game.

All games will be played between one Lonelyville and one Fardistanton player, using tournament rules (which may differ slightly from the standard game of badminton). A game starts with the toss of a coin to see who will serve first. It then consists of a series of rallies. If the player serving wins a rally they get a point, and they continue as server for the next rally. If the player serving loses the rally, no points are awarded, but the other player becomes server for the next rally. The first player to reach 11 points wins the game.

The communicators can send only one bit of information at a time – a zero or a one. The progress of a game is encoded as follows. The first bit sent indicates which side will serve first – a zero if it is a Lonelyville player and a one if it is a Fardistanton player. A bit is then transmitted for each rally – a zero if the server won and a one if the server lost.

Your task is to write a receiving program for this data. For each bit received the person operating your program will type either a 0 or a 1, followed by an <Enter>. Your program should interpret the incoming data and display information on the progress of the game as indicated in the following sample. (Incoming data is underlined).

```
Welcome to the Lonelyville / Fardistanton tournament

Starting Match 1
Enter team to serve first >> 1
Score: Lonelyville  0, Fardistanton  0
Service Fardistanton >> 1
Score: Lonelyville  0, Fardistanton  0
Service Lonelyville  >> 0
Score: Lonelyville  1, Fardistanton  0
Service Lonelyville  >> 0
Score: Lonelyville  2, Fardistanton  0
Service Lonelyville  >> 0
Score: Lonelyville  3, Fardistanton  0
Service Lonelyville  >> 1
Score: Lonelyville  3, Fardistanton  0
Service Fardistanton >> 0
Score: Lonelyville  3, Fardistanton  1
Service Fardistanton >> 1
Score: Lonelyville  3, Fardistanton  1
 . . . (Some lines omitted)
Service Lonelyville  >> 0
Score: Lonelyville 10, Fardistanton  1
Service Lonelyville  >> 0
Match 1 won by Lonelyville

Starting Match 2
 . . .
```

CONTINUED

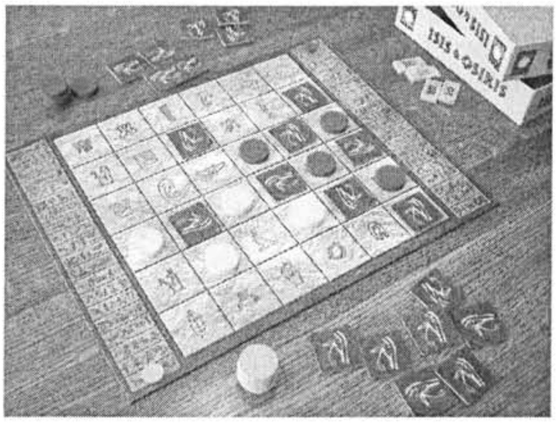
3. **Isis and Osiris (Problem Solving and Programming)**

Your programming work in this question will be assessed on two criteria:

(a) *Your approach to the problem. We will be looking at your work for evidence that you found good ways of storing the necessary data, and devised algorithms for finding and displaying the requested results. Please hand in any notes and diagrams which describe what you are attempting to program, even if you don't have time to code or complete it.*

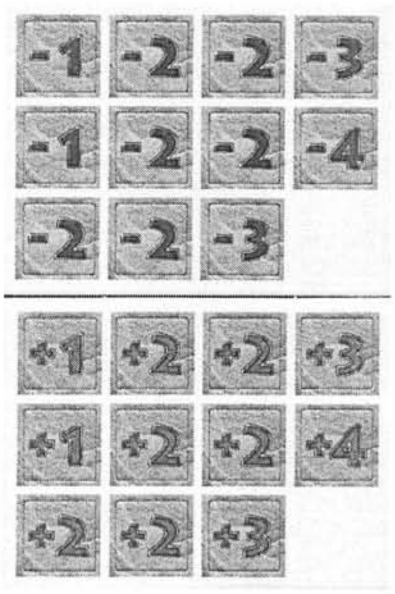
(b) The extent to which your program works and correctly solves the problem.

One part of computer game development (usually called the AI, or Artificial Intelligence, part) is developing a computer strategy, so that the computer can play a good game against human opponents. To test new strategies programmers often play one strategy against another. Your task in this problem is to write a program that can be used to play one strategy against another for the game 'Isis and Osiris'. The simplest possible strategy in most games is 'random'. Whenever it is the computer's turn to play, it selects at random from all plays that are legal at that point in the game. So, 'random' provides a simplest possible strategy that can be used as a starting point for testing. You are required to implement 'random' for the game.



'Isis and Osiris' is a two person board game. It was devised by Michael Schacht, and is available commercially. Its rules, as explained here have been adapted a little to suit the needs of this problem.

The game is played on a 6 by 6 board. It starts with pieces being dealt to each player. The board starts empty. The players take turns to place pieces on the board. Pieces can only be placed on empty squares. No pieces are ever removed. When both players have had 18 turns, all 36 squares of the board are full and the game ends. It is then scored to determine the winner. Scoring depends on the pieces placed. The picture on the left shows the board part way through a game.



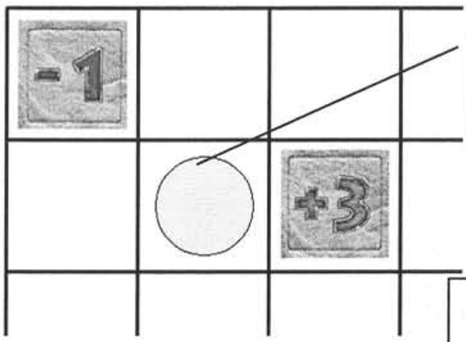
There are two kinds of pieces: stones and tiles. Stones are circular and coloured. One player's stones are blue and the other's yellow. We will refer to the players as 'Blue' and 'Yellow' from now on. Each player starts the game with 8 of their own coloured stones. Tiles are plain coloured squares. Each has a number on it. The numbers range in value from -4 to +4. The game has 22 tiles, shown in the picture on the right: one each of -4 and +4; two each of -3 and +3; six each of -2 and +2; and two each of -1 and + 1. The colours of the numbers on the tiles are not important. At the start of the game 11 tiles are dealt at random to each player. Notice that this means that each player starts with 19 pieces: 8 stones and 11 tiles. During the game they will only play 18 of them (so one will be left over).

Scoring: To score for yellow, examine each yellow stone in turn. Add the value of tiles above, below, left and right of the stone to yellow's score (i.e. directly beside, not diagonally). To score for blue, examine each blue stones in the same manner. See next page for scoring examples..

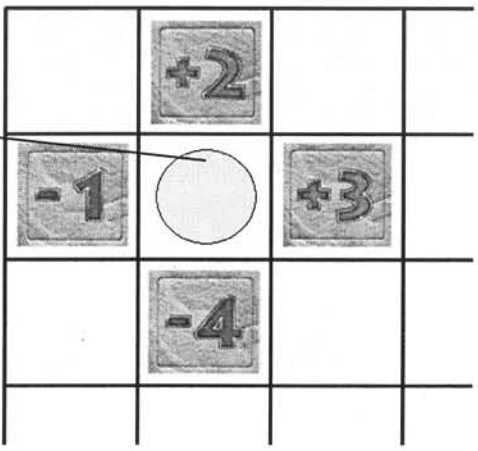
As noted above, there are no restrictions on moves, other than that pieces have to be placed in empty squares. The goal of the game is to place tiles and stones so as to achieve the greatest score. It is good to have tiles with positive numbers beside your stones and tiles with negative numbers beside your opponent's stones. For the purposes of scoring, it doesn't matter which player placed each tile. Yellow always has the first turn.

QUESTION 3 CONTINUED ON NEXT PAGE

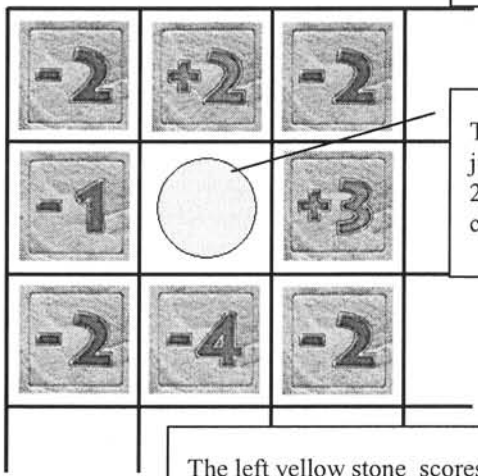
Scoring Examples



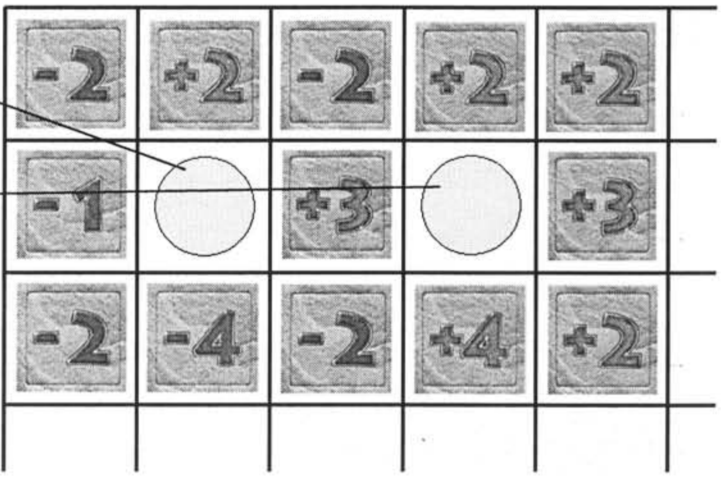
The yellow stone scores +3, from the tile to its right. The -1 tile is not counted



The yellow stone scores 0 (+ 2 + 3 – 1 – 4)

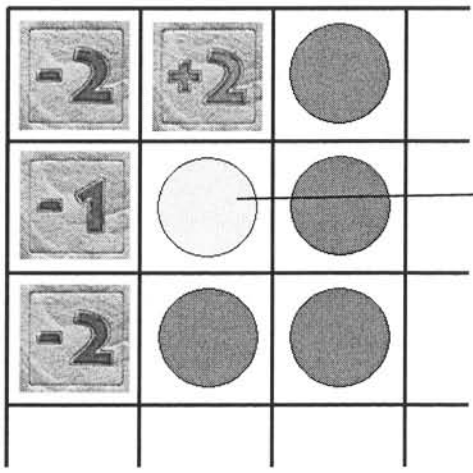


The yellow stone still scores 0, just as for the case above. The -2's in the corners are not counted.



The left yellow stone scores 0 (+ 2 + 3 – 1 – 4)

The right yellow stone scores 12 (+2 + 3 + 3 + 4) Notice that the middle (+3) tile is counted for both yellow stones



The yellow stone scores +1 (+ 2 – 1). The blue stones don't have any effect on the yellow stone's score.

QUESTION 3 CONTINUED ON NEXT PAGE

Your Program

Develop this as a command line program. You do not need to use graphics. A text display of the board will be quite satisfactory. You should be able to show the pieces (stones and tiles) on each square. For example, you might use the letters Y and B to represent yellow and blue stones, and the number on the tile for tiles. You should also show the pieces each player has yet to play.

The program should work as follows:

> When started it should show the board (empty) and the pieces dealt to each player.
>
> It should wait for someone to press <Enter>
>
> It should choose a move for the first player (yellow) and display the board and pieces again
>
> It should wait for <Enter>
>
> Choose and play a move for the second player (blue), displaying the board again.
> (*Remember, the computer is playing both blue and yellow. This program is for testing strategies. You are developing it to the point at which it can play the 'random' strategy for both sides*)
>
> And so on.

Each time your program displays the board it should also display the current score.

The order in which you develop the program is up to you, but we suggest working in the following order.

1. Decide how to store the board and pieces in your program
2. Write instructions to display a board on the screen
3. Write instructions to deal pieces to the players
4. Extend your display to include the pieces yet to be played.
5. Write instructions to choose a move in the 'random' play strategy
6. Write instructions to play a new move each time <Enter> is pressed
7. Add the scoring.

Please hand in printed listings of your programs, and a version of the program on a floppy disk. The version on floppy disk should be a plain text version. You should also hand in any notes, diagrams or explanations you have written. These are particularly important if you do not get your program running correctly as they give us information about your method of approaching the problem. If your program generates output to a file, or to a printer, you should also include printed version of the output.