ENGGEN 131 – Semester Two – 2018

C Programming Project

Deadline: 11:59pm, Friday 19th October
Worth: 12% of your final grade

A note before we begin…

Welcome to the C Programming project for ENGGEN131 2018!

This project is organized around ten tasks. For each task there is a problem description, and you must implement *one function* to solve that problem. You may, of course, define *other functions* which these required functions call upon (i.e. *helper* functions).

Do your very best, but don't worry if you cannot complete every task. You will get credit for each task that you do solve (and you may get partial credit for tasks solved partially).

IMPORTANT - Read carefully
This project is an assessment for the ENGGEN131 course. It is an **individual** project. You do not need to complete all of the tasks, but the tasks you do complete should be an accurate reflection of your capability. You may discuss ideas in general with other students, but writing code must be done by yourself. *No exceptions.* You must not give any other student a copy of your code in any form – and you must not receive code from any other student in any form. There are absolutely NO EXCEPTIONS to this rule.

Please follow this advice while working on this project – the penalties for plagiarism (which include your name being recorded on the misconduct register for the duration of your degree, and/or a period of suspension from Engineering) are simply not worth the risk.

Acceptable	***Unacceptable***
• Describing problems you are having to someone else, either in person or on Piazza, without revealing *any* code you have written • Asking for advice on how to solve a problem, where the advice received is general in nature and does not include any code • Discussing with a friend, away from a computer, ideas or general approaches for the algorithms that you plan to implement (but not working on the code together) • Drawing diagrams that are illustrative of the approach you are planning to take to solve a particular problem (but not writing source code with someone else)	• Working at a computer with another student • Writing code on paper or at a computer, and sharing that code in any way with anyone else • Giving or receiving any amount of code from anyone else in any form • Code sharing = NO

The rules are simple - write the code yourself!

OK, now, on with the project…

Introduction

Connect Four is a strategy game where two players take turns placing tokens into a board, with the goal of forming a straight line (either horizontal, vertical or diagonal) of at least four tokens in a row. Connect Four was first sold by the Milton Bradley board game company in 1974.

The ENGGEN131 variant of Connect Four
Typically, players place tokens at the top of the board which fall down to occupy the lowest available space in each column. As Connect Four is usually played on earth, gravity is used to implement this element of the game. For more information on how this works, refer to https://en.wikipedia.org/wiki/Gravity. In a future where colonies on Mars and long-distance space travel are becoming increasingly likely, a different variant of the game is needed. For this project, you will be implementing a version of Connect Four where a token can be placed into any one of the four sides of the board, and the token slides into the board, along the row or column, until it either reaches the opposite side of the board, or collides with an existing token, or collides with a fixed piece on the board.

The empty board
The Connect Four board is square, and seven different sizes are possible - from 4x4 (the smallest) to 10x10 (the largest). In the center of each board is a fixed area that tokens cannot move through. This fixed area consists of either a single fixed piece at the center (for odd-sized boards) or four fixed pieces at the center (for even-sized boards). Two different empty board configurations are shown below (the '`.`' character signifies an empty space, and the '#' character denotes the fixed central pieces):

```
......
......
..##..
..##..
......
......
```
	```
	.........
	.........
	.........
	.........
	....#....
	.........
	.........
	.........
	.........
	```

***A 6x6 board**. As this is an even-sized board, there are four central "fixed" pieces*	***A 9x9 board**. As this is an odd-sized board, there is a single central "fixed" piece*

Labelling the sides, rows and columns
Each side, row and column of the board can be uniquely identified with a labelling scheme that consist of one character and one digit. The character identifies the *side* of the board, and the digit identifies the *row* or *column*. Two examples are shown below, where the labels are printed around the outside of the board:

```
--NNNNNN--
--012345--
W0......0E
W1......1E
W2..##..2E
W3..##..3E
W4......4E
W5......5E
--012345--
--SSSSSS--
```

A 6x6 board with position labels.

```
--NNNNNNNNN--
--012345678--
W0.........0E
W1.........1E
W2.........2E
W3.........3E
W4....#....4E
W5.........5E
W6.........6E
W7.........7E
W8.........8E
--012345678--
--SSSSSSSSS--
```

A 9x9 board with position labels.

In the examples above, you can see that each side of the board is indicated by a compass direction ('N' = North, 'W' = West, 'E' = East, 'S' = South), and each row and column is indicated by an integer value between 0 and *size*-1 (where *size* is the size of the board). Note that the row and column positions increase going left to right, and top to bottom.

Game play
The first player places a token (either ‘X’ or ‘O’). The second player places a token of the opposite type, and then play continues by alternating between the two players. Each player gets a *single opportunity* to select the position for their token on their turn - if a player (unwisely) selects a position to place their token which is already occupied by an existing token, then their token will not be placed (i.e. they forfeit their turn).

Winning the game
There are two ways to win the game:

1) *Four-in-a-row*: place a token such that a line of at least four tokens is formed (in any direction - vertical, horizontal or diagonal).
2) *Last-to-place*: place the *last* token into the board, such that your opponent is left with no free positions in which to place their token (note: it isn't possible to determine who the last player will be based on who started the game, as some internal spaces on the board may be left unfilled depending on how the tokens are dropped)

Note that these rules for winning the game mean that a draw is not possible - there will always be one winner.

Example

An example of the game being played is shown below. Note that each selected move is indicated with a ***single character*** for the side, followed by a ***single digit*** for the row or column.

1) The start of the game - an empty 7x7 board:

```
--NNNNNNN--
--0123456--
W0.......0E
W1.......1E
W2.......2E
W3...#...3E
W4.......4E
W5.......5E
W6.......6E
--0123456--
--SSSSSSS--
```

2) The first move of the game - the player selects **E3**, with the token coming to rest on the fixed piece

```
--NNNNNNN--
--0123456--
W0.......0E
W1.......1E
W2.......2E
W3...#X..3E
W4.......4E
W5.......5E
W6.......6E
--0123456--
--SSSSSSS--
```

3) The second player responds with **N4**, with their token stopping when it hits the first player's token

```
--NNNNNNN--
--0123456--
W0.......0E
W1.......1E
W2....O..2E
W3...#X..3E
W4.......4E
W5.......5E
W6.......6E
--0123456--
--SSSSSSS--
```

4) Next move: **S4**

```
--NNNNNNN--
--0123456--
W0.......0E
W1.......1E
W2....O..2E
W3...#X..3E
W4....X..4E
W5.......5E
W6.......6E
--0123456--
--SSSSSSS--
```

5) Next move: **W2**

```
--NNNNNNN--
--0123456--
W0.......0E
W1.......1E
W2...OO..2E
W3...#X..3E
W4....X..4E
W5.......5E
W6.......6E
--0123456--
--SSSSSSS--
```

6) Next move: **E2**

```
--NNNNNNN--
--0123456--
W0.......0E
W1.......1E
W2...OOX.2E
W3...#X..3E
W4....X..4E
W5.......5E
W6.......6E
--0123456--
--SSSSSSS--
```

7) Next move: **E3**

```
--NNNNNNN--
--0123456--
W0.......0E
W1.......1E
W2...OOX.2E
W3...#XO.3E
W4....X..4E
W5.......5E
W6.......6E
--0123456--
--SSSSSSS--
```

8) Next move: **S5**

```
--NNNNNNN--
--0123456--
W0.......0E
W1.......1E
W2...OOX.2E
W3...#XO.3E
W4....XX.4E
W5.......5E
W6.......6E
--0123456--
--SSSSSSS--
```

9) Next move: **N3**

```
--NNNNNNN--
--0123456--
W0.......0E
W1...O...1E
W2...OOX.2E
W3...#XO.3E
W4....XX.4E
W5.......5E
W6.......6E
--0123456--
--SSSSSSS--
```

10) Next move: **W4**

```
--NNNNNNN--
--0123456--
W0.......0E
W1...O...1E
W2...OOX.2E
W3...#XO.3E
W4...XXX.4E
W5.......5E
W6.......6E
--0123456--
--SSSSSSS--
```

11) Next move: **E4**

```
--NNNNNNN--
--0123456--
W0.......0E
W1...O...1E
W2...OOX.2E
W3...#XO.3E
W4...XXXO4E
W5.......5E
W6.......6E
--0123456--
--SSSSSSS--
```

Game over!

The last token placed into E4 has generated a diagonal four in a row.

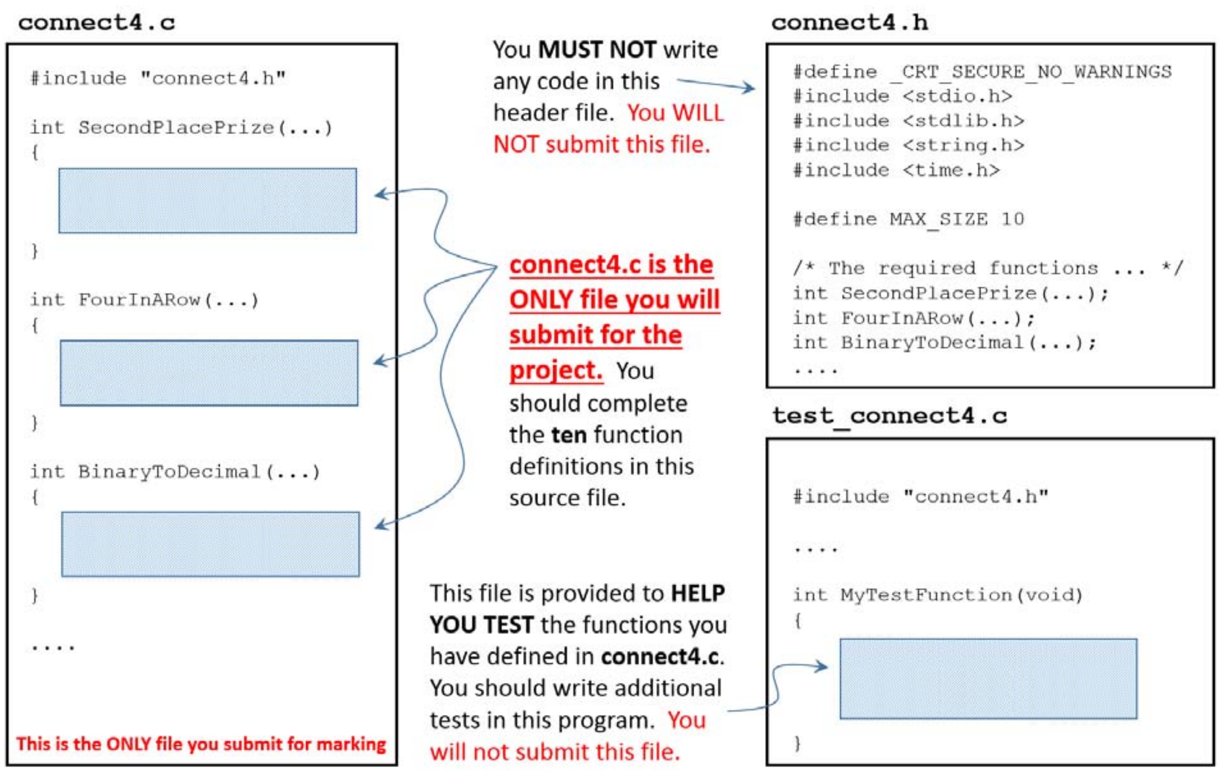
Understanding the project files

There are *three files* that you will be working with: **connect4.c**, **connect4.h** and **test_connect4.c**.

The most important of these three files is the source file **connect4.c**. This is the **ONLY** file that **you will submit for marking**. Please note the following:

- **connect4.c** is a source file that **ONLY CONTAINS FUNCTION DEFINITIONS**
- there is no **main()** function defined in **connect4.c** (and you **must not** add one)
- a separate program, **test_connect4.c**, contains the **main()** function and you can use this to help you test the function definitions that you write in **connect4.c**

The diagram below illustrates the relationship between the three files.



```
#include "connect4.h"

int SecondPlacePrize(...)
{

}

int FourInARow(...)
{

}

int BinaryToDecimal(...)
{

}

....
```

This is the ONLY file you submit for marking

You **MUST NOT** write any code in this header file. You WILL NOT submit this file.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define MAX_SIZE 10

/* The required functions ... */
int SecondPlacePrize(...);
int FourInARow(...);
int BinaryToDecimal(...);
....
```

connect4.c is the ONLY file you will submit for the project. You should complete the **ten** function definitions in this source file.

This file is provided to **HELP YOU TEST** the functions you have defined in **connect4.c**. You should write additional tests in this program. You will not submit this file.

```
#include "connect4.h"

....

int MyTestFunction(void)
{

}
```

The blue shaded regions in the above diagram indicate where you should write code when you are working on the project. There are three simple rules to keep in mind:

- You MUST NOT write any code in **connect4.h** (the header file)
- You MUST write implementations for the functions defined in **connect4.c**
- You SHOULD write additional test code in **test_connect4.c** to thoroughly test the code you write in **connect4.c**

Getting started

To begin, download the file called **CProjectResources.zip** from Canvas. There are three files in this archive:

connect4.c	This is the source file that you will ultimately submit for marking. In this source file you will find the ten functions that you should complete. Initially each function contains an *incorrect* implementation which you should *delete* and then correct. You may add other functions to this source file as you need. **You must not place a main() function in this source file**. This is the only file that you will submit for marking.
connect4.h	This is the header file that contains the prototype declarations for the ten functions you have to write. You must not edit this header file in any way. Both source files (**connect4.c** and **test_connect4.c**) include this header file, and the automated marking program will use the provided definition of **connect4.h**. Modifying this header file in any way will cause an error.
test_connect4.c	This is the source file that contains the **main()** function. This file has been provided to you to help you test the functions that you write. In this file, you should create some example inputs and then call the functions that you have defined inside the **connect4.c** source file. Some simple examples have been included in this file to show you how this can be done.

You might like to start by looking at the **connect4.c** source file. In this source file you will find ten function definitions, however initially they are all implemented *incorrectly*. The prototype declarations are as follows (and these declarations are defined in the **connect4.h** header file):

```
int SecondPlacePrize(int prize1, int prize2, int prize3);
int FourInARow(int values[], int length);
int BinaryToDecimal(int binary);
double MedianAbility(double abilities[], int length);
void RemoveSpaces(char *name);
void InitialiseBoard(int board[MAX_SIZE][MAX_SIZE], int size);
void AddMoveToBoard(int board[MAX_SIZE][MAX_SIZE], int size, char side,
                            int move, int player, int *lastRow, int *lastCol);
int CheckGameOver(int board[MAX_SIZE][MAX_SIZE], int size, int player,
                                                            int row, int col);
void GetDisplayBoardString(int board[MAX_SIZE][MAX_SIZE], int size,
                                                           char *boardString);
void GetMoveBot1(int board[MAX_SIZE][MAX_SIZE], int size, int player,
                                                       char *side, int *move);
void GetMoveBot2(int board[MAX_SIZE][MAX_SIZE], int size, int player,
                                                       char *side, int *move);
```

[Note: GetMoveBot1 and GetMoveBot2 could be defined in the same way, as described later, and so this counts as one function, so there are 10 *different* functions above]

You need to modify and correct the definitions of these functions in **connect4.c**.

Run the test program
You should run the program provided to you in **test_connect4.c**. To do this, start by placing the three source files into an empty folder. You will need to compile both source files. For example, from the Visual Studio Developer Command Prompt, you could type:

```
cl /W4 connect4.c test_connect4.c
```

Or, simply:

```
cl /W4 *.c
```

You should see no warning messages generated when the code compiles.

If you run the program, you should see the following output:

```
ENGGEN131 Project - Semester Two - 2018
                                 _     __
                                | |   / _|
  ___ ___  _ __  _ __   ___  ___| |_  | |_ ___  _   _ _ __
 / __/ _ \| '_ \| '_ \ / _ \/ __| __| |  _/ _ \| | | | '__|
| (_| (_) | | | | | | |  __/ (__| |_  | || (_) | |_| | |
 \___\___/|_| |_|_| |_|\___|\___|\__| |_| \___/ \__,_|_|

Would you like to:

  0 = Test individual functions in connect.c
      [you should write your own test code for this in MyTestFunction()]
  1 = Play Connect Four
      [this will only work if you to have completed all of the functions)]

Please enter your choice [0 or 1]:
```

If you choose option 0, then the function **MyTestFunction()** will be called. This will execute all of the example code in this handout. You can compare the output generated by your functions with the expected output as listed in this handout.

If you choose option 1, then you can play a complete game or a tournament of many games (for example, to test the performance of your bots - refer to more information in the Task Ten description later). In order for this to work however, you will need to have completed the definitions of all of the required functions.

Therefore, it is best to select option 0 until you are confident that your function definitions are correct.

What to submit

You **must not** modify **connect4.h**, although you can modify **test_connect.c**. You will not be submitting either of these files.

You must only submit ONE source file – **connect4.c** – for this project. This source file will be marked by a separate automated marking program which will call your functions with many different inputs and check that they produce the correct outputs.

Testing

Part of the challenge of this project is to **test your functions carefully** with a range of different inputs. It is very important that your functions will never cause the marking program to crash or freeze regardless of the input. If the marking program halts, you cannot earn any marks for the corresponding function. There are three common scenarios that will cause the program to crash and which you must avoid:

- Dividing by zero
- Accessing memory that you shouldn't (such as invalid array indices)
- Infinite loops

Using functions from the standard library

The **connect4.h** header file already includes <stdio.h>, <stdlib.h>, <time.h> and <string.h>. You may not use any other functions from the standard library. If you want some functionality, you must code it!

Marking

The correctness of your project will be marked by a program that calls the functions you have defined in **connect4.c** with many different input values. This program will check that your function definitions return the expected outputs for a large variety of possible inputs. Your mark for the project will essentially be the total number of these tests that are successful, across all ten tasks.

Some tasks are harder than others. If you are unable to complete a task, that is fine – just complete the tasks that you are able to. However, please do not delete any of the ten functions from the **connect4.c** source file. You can simply leave the initial code in the function definition if you choose not to implement it. All ten required functions must be present in the **connect4.c** file you submit for marking (so that the automated marking program is able to compile your code).

Never crash

There is one thing that you must pay important attention to. Your functions must never cause the marking program to crash. If they do, your will forfeit the marks for that task. This is your responsibility to check. There are three common situations that you must avoid:

- Never divide by zero
- Never access any memory location that you shouldn't (such as an invalid array access)
- Never have an infinite loop that causes the program to halt

You must guard against these **very carefully** – regardless of the input values that are passed to your functions. Think very carefully about every array access that you make. In particular, a common error is forgetting to initialise a variable (in which case it will store a "garbage" value), and then using that variable to access a particular index of an array. You cannot be sure what the "garbage" value will be, and it may cause the program to crash.

Array allocation

If you need to declare an array in any of your function definitions, you can make use of this constant from **connect4.h**:

```
#define MAX_SIZE 10
```

You must use a constant value when declaring an array - do not use a variable to declare the size of an array - this will not compile on MSVC. **Please see page 38 of the coursebook**.

Comments

You should include comments within the body of your function definitions, where you feel it is appropriate. If there is a short block of code which is complex, or otherwise it is not immediately obvious what the code does, a short comment at the top of that block can be useful. It will help someone reading your code to understand what the purpose of the code is. However, you should not over-comment your code - this just makes it hard to read. You certainly do not want to have a comment for every line of source code!

If you have used good, meaningful variable names, then much of your code will not need commenting. Have a look at the **test_connect4.c** source file provided to you, to get a rough idea of the amount of commenting that is appropriate. If you look at the PlayConnectFour() function in that file, you will see that there are only a few comments, and each one describes the purpose of a small block of code.

Good luck!

Task One: "Runner up" (10 marks)

You have spent months organising a big Connect Four tournament. To secure funding for prize money, you have spoken to more sponsors than you can even remember. Fortunately, three different companies have agreed to donate money for prizes. You have therefore decided to offer a 1st prize, a 2nd prize and a 3rd prize. The money donated by each company will be used for one of these three prizes. The 1st prize should be the *greatest* amount donated by any one company. The 3rd prize should be the *least* amount donated by any one company.

You have now been asked to report how much the 2nd prize will be. To help you with this seemingly insurmountably difficult task, you have decided to write a function to calculate the value of the 2nd place prize. The inputs to the function will be the three amounts (these will be integers) donated by each of the three sponsoring companies. These amounts may be specified in any order. The output of your function should be the value of the 2nd place prize. Note, if two or more companies have chosen to donate the same amount, then the 2nd place prize might be the same as the 1st or 3rd place prize.

For this task, you must write a function which takes three input integers and returns the middle value (i.e. the 2nd place prize). This function should be called **SecondPlacePrize()**.

Function prototype declaration:

```
int SecondPlacePrize(int prize1, int prize2, int prize3)
```

Assumptions:

You can assume each of the three inputs is a positive integer, greater than zero.

Example:

```
printf("Prize = %d\n", SecondPlacePrize(200, 100, 200));
printf("Prize = %d\n", SecondPlacePrize(45, 600, 590));
```

Expected output:

```
Prize = 200
Prize = 590
```

Task Two: "Four in a row" (10 marks)

Now that you have the prize money secured, you need to start thinking about writing some software to support the Connect Four competition. An obvious place to start would be to have some way to test for four-in-a-row.

Write a function called **FourInARow**() which takes two inputs: an array of integers and the length of the array (i.e. the number of elements in the array that should be examined). The function should return the *index* position of the *first* element in the array that *begins* a four-in-a-row sequence (of any value). If the array does not contain a four-in-a-row sequence, then the function should return -1.

Function prototype declaration:

```
int FourInARow(int values[], int length)
```

Assumptions:

You can assume the length of the array will be at least 1.

Example:

```
int valuesA[10] = {1,2,1,2,1,1,1,1,2,1};
int valuesB[15] = {1,2,1,2,1,1,1,2,2,1,1,4,4,4,4};
int valuesC[15] = {5,6,7,8,8,8,8,8,8,8,8,8,8,8,8};

int result;

result = FourInARow(valuesA, 10);
printf("Result = %d\n", result);

result = FourInARow(valuesB, 15);
printf("Result = %d\n", result);

result = FourInARow(valuesC, 15);
printf("Result = %d\n", result);

result = FourInARow(valuesC, 6);
printf("Result = %d\n", result);
```

NOTE: In this case, because we are only examining the first 6 elements of the array, the function should return -1 (because there is no four-in-a-row sequence amongst the first 6 elements.

Expected output:

```
Result = 4
Result = 11
Result = 3
Result = -1
```

Task Three: “Binary registration” (10 marks)

Hundreds of players have now registered for the Connect Four tournament, and each player has been assigned a unique registration number. However, your assistant (who is a big fan of base 2, I mean, who isn’t?) has insisted on generating registration numbers in binary. To store the registration information in your database, you need to convert all of the binary registration numbers to decimal. To help with this, you need to define a function to convert a binary number to decimal.

Define a function called **BinaryToDecimal**(). The input to your function will be an integer, consisting only of the digits 0 and 1, representing a binary number. The output from your function should be the equivalent decimal value.

Function prototype declaration:

```
int BinaryToDecimal(int binary)
```

Assumptions:

You can assume that the input integer will be greater than 0, will begin with the digit 1, and will only consist of 0 and 1 digits.

Example:

```
printf("Decimal = %d\n", BinaryToDecimal(101));
printf("Decimal = %d\n", BinaryToDecimal(1));
printf("Decimal = %d\n", BinaryToDecimal(11111111));
printf("Decimal = %d\n", BinaryToDecimal(100000000));
```

Expected output:

```
Decimal = 5
Decimal = 1
Decimal = 255
Decimal = 256
```

Task Four: “Middle player” (10 marks)

Now that you have sorted out the registration problem, you need to generate a schedule for the tournament to determine which players will face off against each other. You decide to split all of the registered players into two groups. One group will contain the lower ranked players, and the other group will contain the higher ranked players. The information you have about each player, from their registration details, includes an estimate of their ability (this estimate is a score between 0 and 10).

To help with this, you need to compute the *median estimated ability score* for all of the players. Define a function called **MedianAbility**(). This function takes two inputs: an array of double values (representing the estimated ability scores) and the length of the array. The output of your function should be the median ability score.

The *median* of a set of values is the number that separates the higher half of the values from the lower half. If there are an odd number of values in the set, then the median is simply the middle value (once the values are sorted). If there are an even number of values in the set, then the median is the average of the two middle values.

Function prototype declaration:

```
double MedianAbility(double abilities[], int length);
```

Assumptions:

You can assume that the length of the array will be at least 1. There is no upper limit to the length of the input array (your function must work for arrays of any size). Each value in the array will be between 0.0 and 10.0 inclusive.

Example:

```
double grpA[5]={4.3,2.1,6.7,5.0,1.2};
double grpB[10]={4.3,2.1,6.7,5.0,1.2,9.9,7.0,3.0,6.6,6.5};

double medianAbilityA, medianAbilityB;

medianAbilityA = MedianAbility(grpA, 5);
medianAbilityB = MedianAbility(grpB, 10);

printf("Median ability Group A = %.2f\n", medianAbilityA);
printf("Median ability Group B = %.2f\n", medianAbilityB);
```

Expected output:

```
Median ability Group A = 4.30
Median ability Group B = 5.75
```

Task Five: "Cheaper signs" (10 marks)

You have been asked to help prepare signs that fans can hold when they are watching the Connect Four games to cheer on and support their favourite players. Fans have submitted the text that they want to appear on the signs, and you must now have the signs printed. The cost for printing a sign depends on the number of characters that need to be printed (somewhat surprisingly, this includes space characters). You never quite worked out why a space character should cost the same to print on a sign as other characters, but it is what it is. Although you explored a number of options, you weren't able to secure advertising or any other source of funding to help pay for the signs.

To save money, you have decided to remove some of the unnecessary space characters from the text for each sign. To help with this, you should define a function which takes a string as input, and which modifies the string by removing these unnecessary space characters. If there is more than one space character in a row, anywhere in the string, you want to collapse these so that just a single space character remains. Define a function called **RemoveSpaces**() which is passed a string as input. You should modify the string by removing space characters such that no two space characters remain in a row.

Function prototype declaration:

```
void RemoveSpaces(char *name);
```

Assumptions:

You can assume the length of the string will be at least 1. You **cannot assume an upper length** for the string. Your function should work with strings of any length. This means you cannot define a temporary array to store the resulting string (as you don't know how much space to allocate). See the note on "Array allocation" above. (HINT: a helper function, that removes the character at a given index position, might be very useful!)

Example:

```
char nameA[100] = "The       Champ";
char nameB[100] = "     I    AM      THE     GREATEST    ";
char nameC[100] = "Therearenospaceshereatall";

RemoveSpaces(nameA);
RemoveSpaces(nameB);
RemoveSpaces(nameC);

printf("Name A = [%s]\n", nameA);
printf("Name B = [%s]\n", nameB);
printf("Name C = [%s]\n", nameC);
```

NOTE: In the example for `nameB`, there should still remain one space character at the start and end of the string.

Expected output:

```
Name A = [The Champ]
Name B = [ I AM THE GREATEST ]
Name C = [Therearenospaceshereatall]
```

Task Six: "A Connect Four Board" (10 marks)

A Connect Four board can be represented using an underlying 2-dimensional array of integers. The constant `MAX_SIZE` has been provided to you, and indicates the largest possible dimensions of the Connect Four board.

The following code creates a 10x10 2-dimensional array, and sets each element of the array to -1 (this value has been chosen as it is different to the value that you should use to initialise the array, so the output below is easier to interpret). A nested loop is then used to print out all of these values:

```
int board[MAX_SIZE][MAX_SIZE];
int i, j;
for (i = 0; i < MAX_SIZE; i++) {
      for (j = 0; j < MAX_SIZE; j++) {
            board[i][j] = -1;
      }
}

for (i = 0; i < MAX_SIZE; i++) {
      for (j = 0; j < MAX_SIZE; j++) {
            printf("%d ", board[i][j]);
      }
      printf("\n");
}
```

The output from this code above is:

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

We need a function to initialise the Connect Four board for a given size. This function should set each element of the board to **0** (indicating an empty space) and the middle position (or middle positions for an even sized board) to **3** (indicating a fixed piece). For this task, you must define a function called **InitialiseBoard**() which initialises a provided 2-dimensional array in this way.

Function prototype declaration:

```
void InitialiseBoard(int board[MAX_SIZE][MAX_SIZE], int size);
```

Assumptions:

The value of *size* will be between 4 and 10 inclusive.

For example, consider the code below which initially sets all of the elements of a 2-dimensional array to -1, but then calls the **InitialiseBoard**() function before printing out the array:

```
int board[MAX_SIZE][MAX_SIZE];
int i, j;
for (i = 0; i < MAX_SIZE; i++) {
      for (j = 0; j < MAX_SIZE; j++) {
            board[i][j] = -1;
      }
}

InitialiseBoard(board, 5);

for (i = 0; i < MAX_SIZE; i++) {
      for (j = 0; j < MAX_SIZE; j++) {
            printf("%d ", board[i][j]);
      }
      printf("\n");
}
```

This time, the output would be:

```
0 0 0 0 0 -1 -1 -1 -1 -1
0 0 0 0 0 -1 -1 -1 -1 -1
0 0 3 0 0 -1 -1 -1 -1 -1
0 0 0 0 0 -1 -1 -1 -1 -1
0 0 0 0 0 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

If we initialized the same array for a board of size 8 and then printed the underlying array:

```
InitialiseBoard(board, 8);

for (i = 0; i < MAX_SIZE; i++) {
      for (j = 0; j < MAX_SIZE; j++) {
            printf("%d ", board[i][j]);
      }
      printf("\n");
}
```

this time the output would be as follows:

```
  0 1 2 3 4 5 6 7
0 0 0 0 0 0 0 0 0 -1 -1
1 0 0 0 0 0 0 0 0 -1 -1
2 0 0 0 0 0 0 0 0 -1 -1
3 0 0 0 3 3 0 0 0 -1 -1
4 0 0 0 3 3 0 0 0 -1 -1
5 0 0 0 0 0 0 0 0 -1 -1
6 0 0 0 0 0 0 0 0 -1 -1
7 0 0 0 0 0 0 0 0 -1 -1
  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

size = 8

Task Seven: "Place token" (10 marks)

We now need a function to place a token onto the Connect Four board. The inputs to the function will be the underlying 2-dimensional array, the *size* of the board (which will be between 4 and 10 inclusive), the *side* the user wants to drop the token into (which will be a character, and will either be 'N', 'E', 'W' or 'S'), the *row* or *column* the user wants to place the token into (which will be an integer between 0 and *size*-1), which *player* is placing the token (either **1** or **2**), and *two pointers* (which will be used to record the *final position* at which the token comes to rest). For this task, you must define a function called **AddMoveToBoard**().

Function prototype declaration:

```
void AddMoveToBoard(int board[MAX_SIZE][MAX_SIZE],
                              int size, char side, int move, int player,
                                           int *lastRow, int *lastCol);
```

Assumptions:

You can assume that the board has been initialized for the specified size, and that each element of the underlying array is either **0** (a space), **1** (a token from Player 1), **2** (a token from Player 2) or **3** (a fixed piece in the center of the board). You can also assume that *side* will be 'N', 'E', 'W', or 'S', and *move* will be between 0 and *size*-1. The input *player* will either be **1** or **2** (and this is the value you will need to place onto the board). Finally, you can assume that addresses for *lastRow* and *lastCol* are valid.

Once placed, the token should move into the board as far as possible, until it hits either another token, the opposite side of the board, or a fixed piece. You should update the board by placing the token in its final position (the input *player* indicates whether a **1** or a **2** should be placed in this location of the array). You must also store this final position of the placed token into the two pointers (*lastRow* and *lastCol*). If the user tries to place a token where one already exists (which is a poor move, as it means the new token cannot be placed onto the board), then *lastRow* and *lastCol* should both be set to -1 (and the 2-dimensional array representing the board should not change).

For example, let's place a token (as player 1) into position N1 of a newly initialized 8x8 board, and then print the final position of this token as well as the resulting board:

```
int board[MAX_SIZE][MAX_SIZE];
int i, j, rowPosition, colPosition;
int size = 8;

InitialiseBoard(board, size);

AddMoveToBoard(board, size, 'N', 1, 1, &rowPosition, &colPosition);

printf("Token position: row=%d, col=%d\n\n", rowPosition, colPosition);
for (i = 0; i < size; i++) {
      for (j = 0; j < size; j++) {
            printf("%d ", board[i][j]);
      }
      printf("\n");
}
```

Notice that, in this example, we have included a **printf()** statement to display the final position of the token on the board. The output below shows the final position of the token and the configuration of the board:

```
Token position: row=7, col=1

0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 3 3 0 0 0
0 0 0 3 3 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
```

In the following example, we place three tokens (as alternating players) onto a newly initialized 8x8 board, into positions N1, E4 and S5:

```
int board[MAX_SIZE][MAX_SIZE];
int i, j, rowPosition, colPosition;
int size = 8;

InitialiseBoard(board, size);

AddMoveToBoard(board, size, 'N', 1, 1, &rowPosition, &colPosition);
printf("Token position: row=%d, col=%d\n", rowPosition, colPosition);

AddMoveToBoard(board, size, 'E', 4, 2, &rowPosition, &colPosition);
printf("Token position: row=%d, col=%d\n", rowPosition, colPosition);

AddMoveToBoard(board, size, 'S', 5, 1, &rowPosition, &colPosition);
printf("Token position: row=%d, col=%d\n\n", rowPosition, colPosition);

for (i = 0; i < size; i++) {
      for (j = 0; j < size; j++) {
            printf("%d ", board[i][j]);
      }
      printf("\n");
}
```

This produces the following output:

```
Token position: row=7, col=1
Token position: row=4, col=5
Token position: row=5, col=5

0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 3 3 0 0 0
0 0 0 3 3 2 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
```

Task Eight: "Game over, man!" (10 marks)

We now need to check if the game is over. There are two conditions that need to be checked separately:

1) If the board contains no available positions in which a new token can be placed, then the game is over and the winner is the current player (who must have made the last move).
2) If the last token placed forms four-in-a-row, then the game is over and the winner is the current player (who must have made the last move)

Define a **CheckGameOver**() function which tests to see whether the last move made has ended the game. When this function is called, you are provided with the current *board* configuration (and *size*), as well as the current *player*, and the *row* and *column* position of where the last token placed ended up on the board. The function should return 0 (i.e. false) if the game is *not over*, otherwise it should return the value of *player* (that is, the current player, who placed the last token and must have won the game). This indicates that the current player has won the game.

<u>Function prototype declaration:</u>

```
int CheckGameOver(int board[MAX_SIZE][MAX_SIZE], int size,
                                          int player, int row, int col);
```

<u>Assumptions:</u>

You can assume that the inputs *row* and *col* will store the position in which the last placed token ended up on the board.

In the following example, we begin by initializing a 5x5 board, and then the two players take alternating turns placing tokens into N0 and N1:

```
int board[MAX_SIZE][MAX_SIZE];
int i, j, rowPos, colPos;
int size = 5;
int result;

InitialiseBoard(board, size);

AddMoveToBoard(board, size, 'N', 0, 1, &rowPos, &colPos);
AddMoveToBoard(board, size, 'N', 1, 2, &rowPos, &colPos);

AddMoveToBoard(board, size, 'N', 0, 1, &rowPos, &colPos);
AddMoveToBoard(board, size, 'N', 1, 2, &rowPos, &colPos);

AddMoveToBoard(board, size, 'N', 0, 1, &rowPos, &colPos);
AddMoveToBoard(board, size, 'N', 1, 2, &rowPos, &colPos);
```

If we print the board after these six moves have been made:

```
      for (i = 0; i < size; i++) {
            for (j = 0; j < size; j++) {
                  printf("%d ", board[i][j]);
            }
            printf("\n");
      }
```

we will see the following output:

```
      0 0 0 0 0
      0 0 0 0 0
      1 2 3 0 0
      1 2 0 0 0
      1 2 0 0 0
```

Given this board configuration, if we check to see if Player 2 (who made the last move into *rowPos* and *colPos*) has won:

```
      printf("Checking for win for Player 2 in [%d, %d]\n", rowPos, colPos);
      result = CheckGameOver(board, size, 2, rowPos, colPos);
      printf("Result = %d\n", result);
```

we will see the following output (indicating that they haven't won):

```
      Checking for win for Player 2 in [2, 1]
      Result = 0
```

If we now make one more move (Player 1 this time), print the board again, and check for a win:

```
      AddMoveToBoard(board, size, 'N', 0, 1, &rowPos, &colPos);

      for (i = 0; i < size; i++) {
            for (j = 0; j < size; j++) {
                  printf("%d ", board[i][j]);
            }
            printf("\n");
      }

      printf("Checking for win for Player 1 in [%d, %d]\n", rowPos, colPos);
      result = CheckGameOver(board, size, 1, rowPos, colPos);
      printf("Result = %d\n", result);
```

We see that Player 1 has now won the game:

```
      0 0 0 0 0
      1 0 0 0 0
      1 2 3 0 0
      1 2 0 0 0
      1 2 0 0 0
      Checking for win for Player 1 in [1, 0]
      Result = 1
```

Task Nine: "Pretty board" (10 marks)

In all of the examples up until now, we have displayed the board using a nested loop, and we have shown the raw values in the underlying 2-dimensional array. This doesn't look that nice. In this task, you will generate a nicely formatted string representation of the board.

Function prototype declaration:

```
void GetDisplayBoardString(int board[MAX_SIZE][MAX_SIZE],
                                           int size, char *boardString);
```

Assumptions:

> You can assume that *board* is valid, and only contains the values **0** (empty space), **1** (a token from player 1, which should display as 'X'), **2** (a token from player 2, which should display as 'O') and **3** (a fixed piece on the board, which should display as a '#')

Define a function called **GetDisplayBoardString**() which takes the current *board* (and *size*) as input, as well as a *string buffer* (which is just an array of type char). The function should generate a special formatted string representation of the board - but instead of printing this string, the function should write all of the characters of the string into the string buffer.

For example, the code below creates a newly initialized board of size 4 and a string buffer called **boardString**. It then calls the **GetDisplayBoardString()** function to initialise the **boardString** buffer. Finally, it prints out the length of the string in the **boardString** buffer, and it prints out the string itself:

```
int board[MAX_SIZE][MAX_SIZE];
int size = 4;
char boardString[250];

InitialiseBoard(board, size);
GetDisplayBoardString(board, size, boardString);

printf("The length of the string is: %d\n\n", strlen(boardString));
printf("%s", boardString);
```

The output of the code above is:

```
The length of the board string is: 72

--NNNN--
--0123--
W0....0E
W1.##.1E
W2.##.2E
W3....3E
--0123--
--SSSS--
```

Note that the length of the string in the previous output is **72** characters. This is because the board is 4x4, but includes four extra rows and four extra columns of printable characters making 8x8 printable characters (=64) as well as 8 newline characters, one at the end of each line, for a total of 72 characters

Here is one more example with the largest possible board (10x10) - in this case we have added a few tokens to the board before displaying it:

```
int board[MAX_SIZE][MAX_SIZE];
int size = 10;
int rowPos, colPos;
char boardString[250];

InitialiseBoard(board, size);

AddMoveToBoard(board, size, 'N', 0, 1, &rowPos, &colPos);
AddMoveToBoard(board, size, 'N', 1, 2, &rowPos, &colPos);
AddMoveToBoard(board, size, 'N', 0, 1, &rowPos, &colPos);
AddMoveToBoard(board, size, 'N', 1, 2, &rowPos, &colPos);
AddMoveToBoard(board, size, 'W', 4, 1, &rowPos, &colPos);
AddMoveToBoard(board, size, 'W', 4, 2, &rowPos, &colPos);

GetDisplayBoardString(board, size, boardString);

printf("The length of the string is: %d\n\n", strlen(boardString));
printf("%s", boardString);
```

In this case, the output is:

```
The length of the string is: 210

--NNNNNNNNNN--
--0123456789--
W0..........0E
W1..........1E
W2..........2E
W3..........3E
W4..OX##....4E
W5....##....5E
W6..........6E
W7..........7E
W8XO........8E
W9XO........9E
--0123456789--
--SSSSSSSSSS--
```

Note that **210** characters is the longest possible string representation of a board, as the maximum board size is 10x10. With four extra rows and columns of printable characters on each side of the board, there are 14x14 printable characters (=196) as well as 14 newline characters at the end of each line, making a total of 210 characters.

Task Ten: "Bots!" (10 marks)

Release the bots! Playing Connect Four by yourself isn't much fun. In this task you should define the logic for a bot that can play Connect Four (although it doesn't have to play very well!).

For this task, you **must define two bots**. However, if you like, these two bots can use exactly the same strategy (you can use exactly the same code for each of them). But it might be more interesting to experiment with different strategies. That way, you can have the bots play against each other and see which strategy is best!

Function prototype declarations:

```
void GetMoveBot1(int board[MAX_SIZE][MAX_SIZE], int size, int player,
                                                 char *side, int *move);
void GetMoveBot2(int board[MAX_SIZE][MAX_SIZE], int size, int player,
                                                 char *side, int *move);
```

Assumptions:

> You can assume that when these functions are called, the board will not be completely full - that is, there will always be at least one open slot in which the bot can place a token. Why is this? Well, remember that after a player has placed their token, if *that move* fills up the board, then the game is over (and we would not ask for another move to be made).

When one of these functions is called, it is passed the current *board* configuration as input, as well as the *size* of the board, and which *player* the bot is controlling (this could be **1** or **2**). Once the bot has chosen where to place their token, this is indicated by writing the chosen move into the pointers *side* and *move* (which are provided as input to the function).

In the example below, the bots use the same (trivial) strategy of choosing one side at random, and then choosing one position on the side in which to place their token. The following code creates a 9x9 board and then has these two simple bots each make one move:

```
int board[MAX_SIZE][MAX_SIZE];
int size = 9;
int rowPos, colPos, move;
char side;
char boardString[250];

InitialiseBoard(board, size);

// Bot 1 is Player 1:
GetMoveBot1(board, size, 1, &side, &move);
AddMoveToBoard(board, size, side, move, 1, &rowPos, &colPos);

printf("Bot has chosen: side=%c move=%d\n", side, move);
printf("Token is at: row=%d, col=%d\n\n", rowPos, colPos);
```

```
// Bot 2 is Player 2:
GetMoveBot2(board, size, 1, &side, &move);
AddMoveToBoard(board, size, side, move, 2, &rowPos, &colPos);

printf("Bot has chosen: side=%c move=%d\n", side, move);
printf("Token is at: row=%d, col=%d\n\n", rowPos, colPos);

// Display the moves:
GetDisplayBoardString(board, size, boardString);
printf("%s", boardString);
```

The output from this code is as follows:

```
Bot has chosen: side=N move=4
Token is at: row=3, col=4

Bot has chosen: side=W move=2
Token is at: row=2, col=8

--NNNNNNNNN--
--012345678--
W0.........0E
W1.........1E
W2........O2E
W3....X....3E
W4....#....4E
W5.........5E
W6.........6E
W7.........7E
W8.........8E
--012345678--
--SSSSSSSSS--
```

The first bot placed a token at N4 (which landed on the fixed piece in the center of the board), and the second bot placed a token at W2.

Your bots can follow any strategy you like, however to earn the marks for this task, they must meet the following two requirements:

1) the bot must return a **valid move** (see “Assumption” above - there is guaranteed to be at least one available position on the board)
2) if it is possible for the bot to win on its current move, then it must make a **winning move**

Competition
As an optional part of this project, you can choose to take part in a class-wide competition to see which bot is the best. To develop your bot’s strategy, you can have your two bots compete against each other, in a tournament, and see which one performs the best.

When you run the **test_connect4.c** program, you can choose the option to have your two bots compete. This will play the strategy defined in the **GetMoveBot1()** function against the strategy defined in the **GetMoveBot2()** function.

In the example below, the two bots compete in a 100,000 game tournament:

```
Enter board size: 8
Options:
 [1] = Human vs. Human
 [2] = Human vs. Bot1
 [3] = Bot1 vs. Bot2
Choose game type: 3
Number of games: 100000

Tournament over! Games played = 100000
Player 1 wins = 84254 / Player 2 wins = 15746
```

In this case, Bot1 is clearly using a better strategy than Bot2.

Good luck!

BEFORE YOU SUBMIT YOUR PROJECT

Warning messages

You should ensure that there are **no warning messages** produced by the compiler (using the `/W4` option from the VS Developer Command Prompt).

REQUIRED: Compile with Visual Studio before submission

The marking program uses the VS Developer Command Prompt environment.

Even if you haven't completed all of the tasks, your code **must** compile successfully. You will get some credit for a partially completed task if the expected output matches the output produced by your function. **If your code does not compile, your project mark will be 0.**

You may use any modern C environment to develop your solution, however *prior to submission* you must check that your code compiles and runs successfully using the Visual Studio Developer Command Prompt. This is not optional - it is a requirement for you to check this. During marking, if there is an error that is due to the environment you have used, and you failed to check this using the Visual Studio Developer Command Prompt, you will receive 0 marks for the project. Please adhere to this requirement.

In summary, before you submit your work for marking:

STEP 1:	Create an empty folder on disk
STEP 2:	Copy **just** the source files for this project (your completed **connect4.c** file and the unedited **test_connect4.c** and **connect.h** files) into this empty folder
STEP 3:	Open a Visual Studio Developer Command Prompt window (as described in Lab 7) and change the current directory to the folder that contains these files
STEP 4:	Compile the program using the command line tool, with the warning level on 4: `cl /W4 *.c` If there are warnings for code you have written, **you should fix them**. You **should not submit** code that generates **any** warnings.

Do not submit code that does not compile!

And finally…

This project is an **assessed piece of coursework**, and it is essential that the work you submit reflects what you are capable of doing. You **must not copy any source code** for this project and submit it as your own work. You must also **not allow anyone to copy your work**. All submissions for this project will be checked, and any cases of copying/plagiarism will be dealt with severely. We really hope there are no issues this semester in ENGGEN131, as it is a painful process for everyone involved, so please be sensible!

Ask yourself:

have I written the source code for this project myself?

If the answer is "no", then **please talk to us before the projects are marked**.

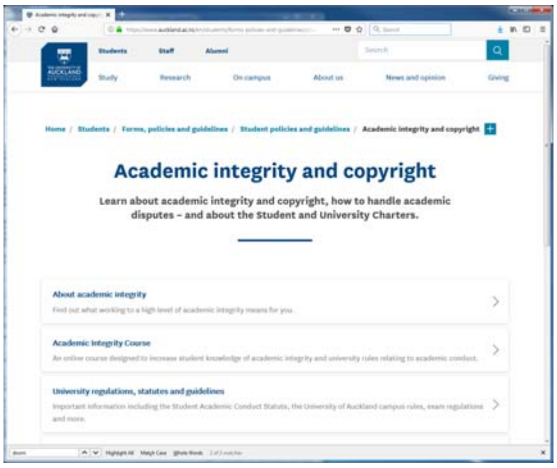
Ask yourself:

have I given anyone access to the source code
that I have written for this project?

If the answer is "yes", then **please talk to us before the projects are marked**.

Once the projects have been marked it is too late. There is more information regarding The University of Auckland's policies on academic honesty and plagiarism here:

`http://www.auckland.ac.nz/uoa/home/about/teaching-learning/honesty`



Home / Students / Forms, policies and guidelines / Student policies and guidelines / Academic integrity and copyright

Academic integrity and copyright

Learn about academic integrity and copyright, how to handle academic disputes – and about the Student and University Charters.

About academic integrity
Find out what working to a high level of academic integrity means for you.

Academic Integrity Course
An online course designed to increase student knowledge of academic integrity and university rules relating to academic conduct.

University regulations, statutes and guidelines
Important information including the Student Academic Conduct Statute, the University of Auckland campus rules, exam regulations and more.

ENGGEN131 C Project 2018

Dr Paul Denny
Computer Science
The University of Auckland
New Zealand
Ph: +64 9 3737599 x 87087