The University of Melbourne
School of Computing and Information Systems
COMP10002 Foundations of Algorithms
Semester 1, 2025
Assignment 1: Exploring Spaces
Due: 11:59 pm Friday 02 May 2025
Version: 1.4.2

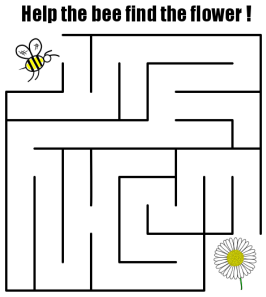
April 28, 2025

IMPORTANT NOTES:

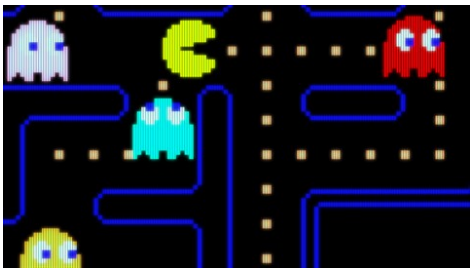
- Clarified where to write bonus challenge code (V1.4.2)
- Please be aware that we have changed the submission location to be on GRADESCOPE, instead of ED. You will need to submit one file `a1.c` onto the grade-scope submission tab.
- Please note that the auto-grader on gradescope will provide suggestions on your coding style that will contribute to the style mark.
- The expectations for Task 2 have been further clarified.
- For the bonus challenge we have made it count for 1 bonus mark

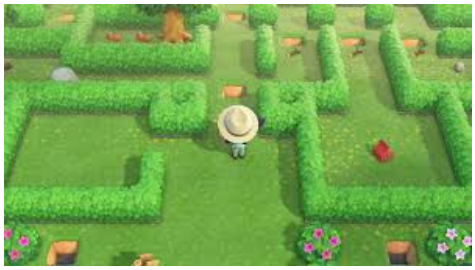
Please see the FAQ for instructions on how to submit the assignment.

At some point in time you may have seen one of these... You have prob-

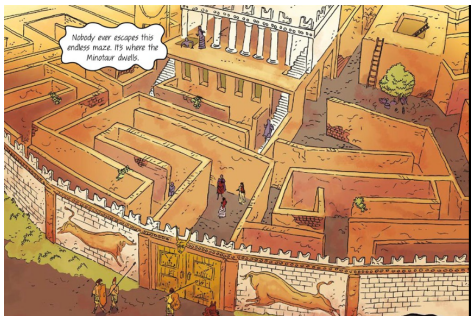


ably even encountered them in a variety of video games from Pac-man to Animal Crossing...





Or maybe you have heard the myths of labyrinths with monsters hidden in their depths...



This is because spatial exploration puzzles, like mazes are one of the oldest form of playful activities we have.

We live a world where we are constantly physically exploring and navigating with our bodies. So it's almost a no-brainer that we would see

this kind of activity show up in our play. Spatial exploration puzzles span many different types of navigation problems, but for today we will focus on one called ***pathfinding***.

Pathfinding is the act of finding a possible path between two locations. Often we are looking for the shortest-path, but sometimes we may look for other types of paths. In modern games, pathfinding is used to tell characters how to move through the world. When people are chasing you in the Untitled Goose Game, or your Sim can't get out of the pool because you deleted the ladder, those are examples of path finding at work.

For this assessment, we are going to be putting our algorithmic thinking to the test by exploring simple pathfinding.

About the Assignment

Assignment Structure

Like a game, this assignment is divided into levels with different tasks. Like any good game, each subsequent level builds on the skills and techniques you used in the previous level. We recommend trying them all in order for this reason.

To help get you started, each level will outline the problem, your requirements for the level, any information about the problem space that may be useful to you, and an example of output from one of the provided test files. Occasionally we will also highlight helpful hints and tips in a blue box like so:

Hey! Listen!

This is an example of what a hint/tip will look like.

Bonus Challenge:

At the end of the assignment, there is a skill-testing bonus challenge. The bonus challenge is a slightly more advanced problem that you are invited to try. There is one bonus mark associated with this.

Learning Outcomes

By completing this assessment you will demonstrate your understanding of the following coding concepts:

- Input processing,
- Control structures (conditionals and loops),
- Functions, and
- Arrays.

You will also be demonstrating your algorithmic problem solving skills, particularly showing how you understand searching problems and recursion.

Getting Started

For this assignment, you will be given the following:

- a skeleton code file (`a1.c`) with a semi-complete main function, a set of function prototypes for you to complete, and comment space for your answers to written questions;
- a set of test input files for the various levels (`test0.txt`, `test1.txt`, `test2.txt`); and,
- examples of correct outputs for the test files both in this document, and as a set of files such as (`test0-level1-out.txt`, `test0-level2-out.txt`, `test0-level3-out.txt`, `test0-level4-out.txt`) etc.

Before you start coding, you need to get to know the structure of the game. After reading this assignment and the FAQ at the end of this document, spend time familiarizing yourself with `(a1.c)`. Pay attention to the variables, constants, and function prototypes already in-place. These will be useful hints at how to complete the functions later!
And remember, *Algorithms are fun!*

Level 1: Hello World? [4 marks]

In order to do any pathfinding, we need a representation of a "world" that we can move around in. But how do we represent that on a computer?

There are a variety of ways, but for simplicity we can think of our world like a map with a grid imposed on top of it. This way every location on the map has an x and y coordinate that tells us where on the map it is.

We can extend this understanding to think about our map as a two-dimensional array. Every element of this 2D array is the relevant structure at that `(x, y)` point. For example, imagine a world where there are walls blocking our path; the array could store a symbol of a wall at the real wall's `(x, y)` location to show us that it is there.

This leads us to **our first task: create the map for our world**!

Requirements

To successfully complete this level you must:

1. Complete the function FillMap() so it can read in the map data from the test file;

   Hey! Listen!

   We are running your code using input redirection (using < in commandline) to pass the test file to program so you can use the standard input functions, like scanf! To have a closer look at the command we are running, see the `BUILD_HELP.txt` file in the skeleton code folder we have provided.

2. Complete a function called `PrintMap()` that prints out the map to the terminal screen; and,

3. Save the starting and ending locations you found to the appropriate variables in the main function.

> **Hey! Listen!**
>
> Think about how this task interacts with C scopes, and what tools we have to manipulate variables in different scopes.

What we know about the problem

File structure. The input file will always contain at least 3 lines of data.

- Line 1:
  - Two positive integers representing the row (`y` coordinate) and column (`x` coordinate) of the STARTING position
- Line 2:
  - Two positive integers representing the row (`y` coordinate) and column (`x` coordinate) of the ENDING position
- Line 3:
  - One positive integer representing the number of obstacles (i.e. BLOCKS) on the map

Depending on the number of blocks, the input file will also have the coordinates for those blocks.

Map size. To make the process easier, we have fixed the map size as a square grid. The specific size value is defined as a preprocessor directive using `#define␣N` at the top of the `(a1.c)` file. During graded testing this number and the input we test on may change. So when creating your code, make sure it can scale with the map size.

Location information. You can assume for this input that we will not put the starting, ending, and/or blocks in the same location. This means the start and end will not be the same place. This will be true for all tests.

Sample Output

Your output for the Level given the same test file, should look like Fig. 1.

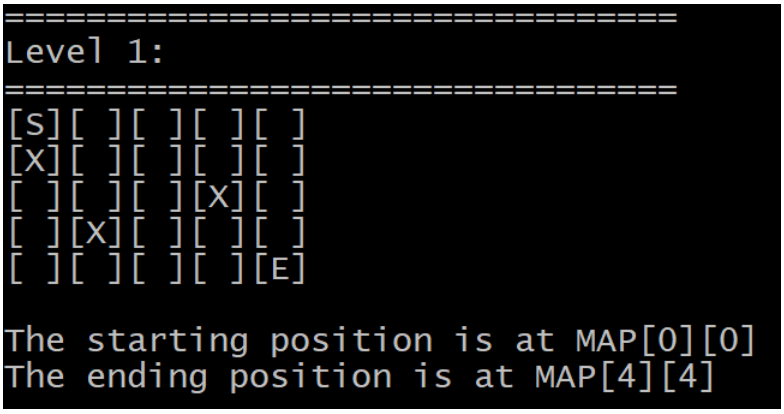


Figure 1: Example of expected output for Level 1 using `test2.txt`

Level 2: Naïve Pathfinding [6 marks]

Great! Now we have our map rendered in the computer and know where the starting and ending points!
We now want to start working towards finding our way from the starting point to the exit point! Let's start by taking a somewhat intuitive approach we will call **naïve pathfinding**.

~~In our naïve pathfinding we will start moving in one direction (e.g. down the rows) until we can't move anymore or until we have reached the correct row for the exit. If we ever reach a block, we will try to move one column over towards the exit, and then continue moving down. Once we have reached the correct row, we will then start moving columns towards the exit using the same logic (going until we hit a block, then moving up or down a row to compensate, before returning to the columns).~~

See requirements for a description of the algorithm below. Fig. 2 renders what the map would look like after each step.

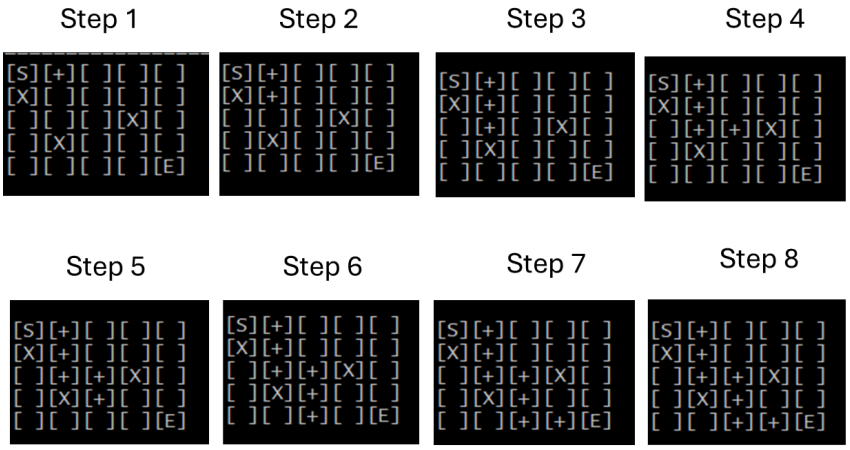


Figure 2: Example of each step the path would take for Level 2 using `test2.txt`

For your **second task**, you will need to implement the described function and answer some questions about it.

Requirements

To successfully complete this level you must:

- Complete the function `SimpleDirections()` so it can both render a path between the starting and ending space using `+`, and output the number of steps needed to reach the end. The pathfinding must use the following logic:
  - From the starting point, the path will always try move along the rows towards the row containing the end point,
  - If the path hits a block, it will move left or right one column in the direction towards the end point,
  - The path will repeat the previous two steps until it reaches the correct row.
  - ~~The path then repeats the above process for the columns (i.e. moving along the columns until it encounters a block, and then moving around the block through the rows)~~
  - The path then begins to move across the columns in the direction of the end point.

  - If at any stage, you cannot make a move following the above rules you must stop.

- Modify the skeleton code, so that in the case that you have no more valid moves from the above algorithm, so that it now prints `"SimpleDirections took N steps and got stuck.\n\n"`, and then the map instead of `"SimpleDirections took N steps to find the goal.\n\n"` and then the map, where N is the number of steps taken before not having any valid moves or reaching the end point.

- Answer the following questions in the comment section indicated at the end of your `a1.c` file:

  - List some cases where this process will not produce our expected result of reaching the end point? Why is this?

    Hey! Listen!

    Think about how the assumptions we have to make about the problem space (i.e. the map).

  - Do you think this process will be efficient for bigger maps? Why, or why not?

What we know about the problem

Assumptions about the map. At this point you can assume the following information:

- ~~There will always be at least one path that will take you from the start to end. This may not always be the case in the future.~~ We have provided an example in Fig. 4 of the expected output.

- There is only one starting and one ending position. However, you should consider that the starting and ending positions could be anywhere on the map grid.

Sample Output

Your output for the Level given the same test file, should look like Fig. 3 or Fig. 4.

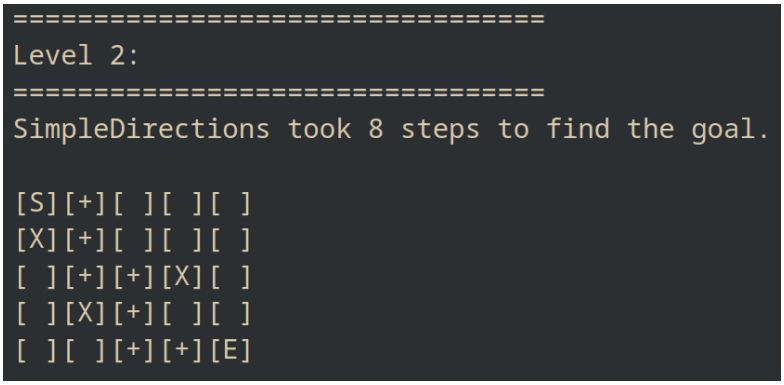


Figure 3: Example of output from Level 2 using `test2.txt` input.

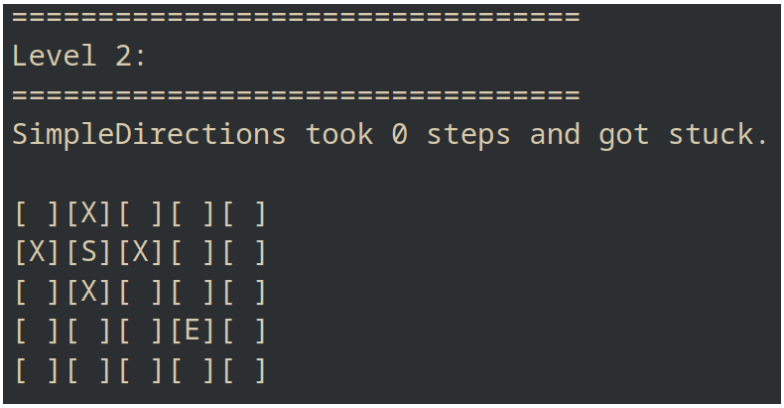


Figure 4: Example of output from Level 2 using `test3.txt` input.

Level 3: Closest Neighbours [4 marks]

Hooray that's one way to find a path to the end! Let's try to come up with another (slightly more generic) process!
Intuitively, we may want to start by finding a valid (i.e. unblocked, empty) space adjacent to our starting space — i.e. its neighbour. If we then move into that new valid space, we can repeat the search process by just looking for the next adjacent empty space. In this way, we are not pre-emptively deciding on the direction of our search, and so we do not need to give our search an end point. We call this process **Closest Neighbours**.
Our third task, is to implement this Closest Neighbours algorithm as discussed. Fig. 5 illustrates what the path looks like on the map at every step of this Closest Neighbours algorithm.

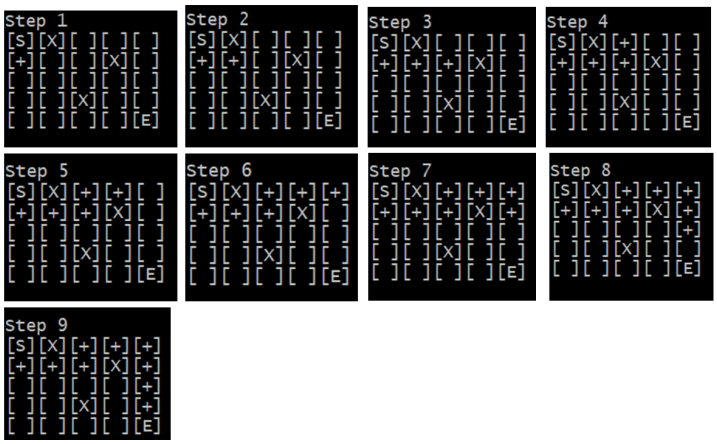


Figure 5: Example of each step the path would take for Level 3 using test1.txt

Requirements

To successfully complete this level you must:

- Complete the function `ClosestFreeNeighbour()` to find a path from a known starting position to an unknown ending position using the following logic:
  - Check if your neighbour is empty, and if yes then move to it
  - We check the neighbours in the following order:
    1. Above
    2. Right
    3. Down
    4. Left
  - We end the function once we find the ending position or there is no available neighbour to move to.

What we know about the problem

Assumptions about neighbours. We know that any individual location will have at most four (4) neighbours: above, right, down, and left. Based on this we should be able to know where a neighbour is relative to our current location.

Structure of the process. We can see that this process requires completing the same process on a smaller version of the same problem. We can leverage this to approach this problem using a particular technique that we covered in lecture.

> **Hey! Listen!**
>
> Think about the Triangles and Tower of Hanoi example from lecture!

Assumptions about Map data. ~~You can assume that the map is back to its original state when you start this process. We have provided you with a `RefreshMap()` function which wipes the map back to just its starting configuration. We already call this for you between every Level() call in the main function so that you do not have to worry about carrying over data.~~
This function is not needed in the current iteration of the program that runs each level with a seperate instance of the program, it does however still exist within the program.

Sample Output

Your output for the Level given the same test file, should look like Fig. 6.

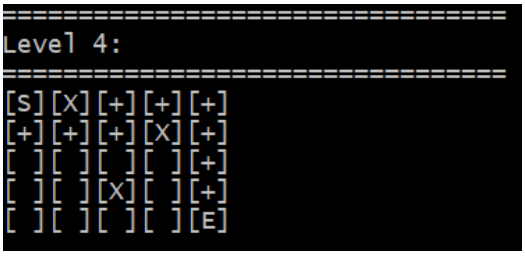


Figure 6: Example of output from Level 3 using `test2.txt` input.

Level 4: Finding a Complete Path [2 marks]

Awesome, now we have some way to find a path from a starting location to an unknown ending point! But will this method always work?

Our final task is to theorize about how we could improve on our Closest Neighbours approach.

Requirements

To successfully complete this level you must answer the following questions in the comment section indicated at the end of your `a1.c` file:

- Do you think the Closest Neighbour process will always work? Why or why not? Write out what you know about the problem space (i.e. possible board setups) to support your thoughts about the process.
- How would you improve the Closest Neighbour process to handle edge cases? Use pseudocode to describe your proposed algorithm.

> **Hey! Listen!**
>
> Think about how we could remember where we have been.

What we know about the problem.

For this final task, you should write out what you know about the problem space of mazes (generically). Thinking about what the problem space looks like, is a very important skill in algorithmic thinking.

Sample Output

Your output for the Level is just text and pseudocode in a comment in your assignment code file.

Bonus Challenge [1 bonus mark]

Note that in the test cases given this is described as `taskN-level-4-out.txt`

Bonus Challenge:

Now that you have proposed a way to improve on the Closest Neighbours method, we need to try it out. **In this Skill Testing Challenge, you must try to implement the changes you suggested in Level 4 so that you have a more robust maze solving algorithm!** Your implementation should mark all paths you visited/tried with a * and the final resulting path with +.

We test the bonus problem by passing in the command line arguments of -level 4 . In the preceding tasks we also pass in the corresponding level and use that to run specific functions in the code. You need to modify the skeleton to work with the 'level 4' input and run a new function you add. You can use the way it is set up for the previous problems as a guide!

If the bonus challenge fails to find a valid path you should output `"No path found\n"` and then the map state. See test3 for an example of this.

FAQ

Here are some frequently asked questions about submission and policies for this assignment.

How do I submit the assignment?

~~You will need to submit your assignment on the EdStem Platform, as you have used for the assessed problem sheets.~~

You will need to submit your assignment on the Canvas LMS using Gradescope. You may write your code outside the Ed platform in a code editor such as Visual Studio if you wish, but you will need to copy your finalised code onto Ed to submit it. To submit your code, you will need to:

1. Log in to LMS subject site,
2. Navigate to "Assignment 1" in the "Assignments" page,
3. Click on "Load Assignment 1 in a new window",

4. Follow the instructions on the Gradescope "Assignment 1" page,
5. Click on the "Submit" link to make a submission.

Deadline and Late Submissions

The deadline for this assignment is **11:59 pm Friday 02 May 2025**. You can submit as many times as you want to before this deadline. *Only the last submission made before the deadline will be marked.* You can (and should) submit both **early and often** – to check that your program compiles correctly on our test system, which may have some different characteristics to your own machines.

Submissions made after the deadline will incur penalty marks at the rate of -3 marks per day or part day late. Late submissions after 11:59 pm Tuesday 06 May 2025 will **not** be accepted. Do *not* submit after the deadline unless a late submission is intended.

Special Considerations

Special 24-hour extension. If you need just a little more time to finish your assignment, we are offering an **automatically approved 24-hour extension to students who fill out this Automatic Extension Form**. There is no further documentation required, and you are guaranteed to be approved for this extension.

1-3 Day Extensions: For extensions between 1 to 3 working days, students may complete the FEIT online declaration form. The Online Declaration Form covers most written assessments, and must be submitted before the assignment due date. The Online Declaration form does not require any medical documentation from students at time of submission, but this could be asked for at a later date. **Extensions applied for through the FEIT system are not automatically approved.**

4+ Day Extension: If you need more than 3 working days, or alternative considerations for an assignment, you will need to submit a request on the Special Consideration Portal. The Special Consideration application will require students to provide supporting documentation for their circumstance. The full application must be submitted within four working days of the assessment due date; it can be submitted in advance of the due date

if you know you will be impacted. **Special Consideration is not automatically granted. They are assessed by Student and Scholarly Services (SASS) and passed to FEIT if you are found eligible for consideration.**

Academic Adjustment Plans: For those of you who have registered your Academic Adjust Plan with our subject, please follow the details set out in your plan regarding any considerations for deadlines, etc.

How will I be marked?

The assignment is out of 20 marks. The breakdown of marks in the assignment is:

- Level 1 [4 marks]
- Level 2 [6 marks]
- Level 3 [4 marks]
- Level 4 [2 marks + 1 bonus mark]
- Code Structure and Style [4 marks]

This assignment has both coding and written components. Below we have outlined how they will be marked.

Marking your code

Overall your code will be marked on its functionality, structure, and style. Details about the specifics for each Level is in the marking rubric. Here we have a brief description of what we mean by functionality, structure, and style.

Functionality. Your code's functionality will be autograded by test cases. Given the size of this class it is the only feasible way for us to mark your work. Part of your code mark for each Level will be based on how many of our test cases it passes.

IMPORTANT !

Since we are autograding your code, it ***MUST*** compile in order to receive a grade. If your code does not compile, you will receive a 0 for the functionality portion of your work. Our markers **will not** be spending any time troubleshooting it for you during marking.

Code Structure. Part of your code mark will be based on the structure of your actual code. The marker will take a quick look at the functions you have created to determine if you are using the appropriate techniques for the problem.

Code Style. Your code style is important to develop as it makes your code easier to read and debug. To this end 4 marks in the assignment is reserved for evaluating your consistency with coding style and good practice behaviours. We are particularly looking for some of the following:

- Appropriate code commenting,
- Consistent and reasonable naming conventions for functions and variables,
- Consistent indentation, bracket placement, whitespace, short lines, and other code readability elements like authorship comment,
- Appropriate use of maintenance and abstraction tools for efficient code design (e.g. `#define` variables, functions use to reuse code)

This list is not comprehensive, and "appropriateness" is up to interpretation by assignment marker based on a holistic look at your code.
To help you develop good style, we suggest taking a look at some of the following style guides:

- CS50 Style Guide,
- Google's Style Guide (Written for C++),
- Linux Kernal Coding Style Guide

You do not have to follow any of these guides specifically, but note their similarities when you are developing your own coding style.

Marking written answers

The goal of the written questions is to make you think critically about the code you just wrote, and whether it is a robust solution to the problem. As such, we are more concerned with seeing that you have genuinely attempted to think about the problem than having a singular correct answer.

To reflect this, written answers are marked based on both the correctness ***and*** quality of an attempt. This means a genuine attempt, even if the result is wrong will still earn you some mark so we encourage you to really try. A good answer attempt should:

- Incorporate the terminology and language from the course;
- When appropriate, refer to the algorithms and concepts you are familiar with from the course (e.g. does your approach look like any other problem solving techniques?)

Working with Friends and Academic Integrity

Learning is an uncomfortable process of going from not knowing something to knowing something. So while our assignments are designed to be achievable, they can also be challenging because they are asking you to demonstrate that learning in an active way. This means there are going to be times in the learning process where you are struggling. We aim to support you with our lectures, workshops and the First Year Centre. But another important form of support are your peers in the class with you!

It is really helpful to have people around you who are also working towards learning the same material. We strongly encourage you to discuss your work with others, but what gets typed into your program must be individual work, **not** from anyone else. If you are copying solutions from a friend or the internet, you lose out on the chance to do the real learning that will help you throughout this subject and beyond.

We know it is hard to watch our friends struggle, especially when we see how much effort they are putting into their work. However, giving them your code removes their opportunity to learn and grow. The best way to help your friends in this regard is to say a very firm "no" when they ask for a copy of, or to see, your program. Feel free to discuss concepts with them, and direct them to one of the other subject supports like their workshop tutors, the First Year Centre, the PASS program, ED Discussion boards, or

the professors. We truly want you all to succeed, and are willing to put in the time and effort to help anyone struggling at any stage of the learning process.

So please, do **not** give (hard or soft) copies of your work to anyone else; do **not** "lend" your memory stick to others; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest".

A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions in "compare every pair" mode. See https://academichonesty.unimelb.edu.au for more information.