**Task 1: Lesson Review**

## Understanding Algorithms, Problem-Solving, and Pseudo-Code

What is an Algorithm?

An algorithm is a precise and well-defined set of instructions, much like a recipe, designed to solve a specific problem. It is a sequence of computational steps that transforms a given input into a desired output. For an algorithm to be considered correct, it must halt with the correct output for every possible input instance. Algorithms are fundamental to computer science and are used to solve a vast range of problems, from sorting a list of numbers to navigating a maze or making a financial decision.

Two Perspectives on Algorithms

1. Bottom-Up Perspective: This traditional computer science view treats an algorithm as a detailed, well-defined computational procedure ready to be translated into executable code. The focus is on the mechanics of the algorithm itself—its efficiency, implementation, and correctness—as a tool for solving a well-specified computational problem.
2. Top-Down Perspective (Problem-Solving): This broader view sees algorithms as just one critical ingredient within the larger discipline of problem-solving. This perspective, rooted in mathematics and other fields, emphasizes a structured approach to tackling problems, often following a four-step process popularized by George Pólya:
    - Understand: Grasp the problem completely.
    - Plan: Devise a strategy, which in computer science is "algorithm design."
    - Solve: Execute the plan.
    - Check: Verify the solution. If it fails, the process is circular, returning to the "Understand" phase.

As a computer scientist, the ultimate goal is to develop the skill of matching the right algorithm to the right problem, which comes from experience and exposure to a wide range of problems and algorithmic techniques.

The Role of Data Structures

A data structure is a method for storing and organizing data to allow for efficient access and modification (e.g., arrays, linked lists, graphs). While algorithms and data structures are closely related, they are distinct. Data structures provide the framework for holding data, while algorithms define the operations that can be performed on that data (e.g., an algorithm is needed to traverse a graph data structure).

Pseudo-Code: Bridging Strategy and Implementation

Pseudo-code is a high-level, human-readable description of an algorithm's operating principle. It is an essential tool for programmers because it serves two main purposes:

1. Clear Communication: It allows developers to discuss and refine a solution without getting bogged down by the strict syntax of a specific programming language.
2. Easy Conversion to Code: A well-written pseudo-code can be easily translated into an actual programming language like Python, C++, or Java.

Forms of Pseudo-Code:

- Good: Can be a "Typical" form using common programming constructs (if/else, while, function calls) or, increasingly, can be written using Python syntax. Python's readability and high-level nature make it an excellent choice for pseudo-code, and it is becoming a de-facto standard.
- Bad: Flow charts are generally discouraged for describing algorithms as they are less flexible and harder to translate into code.

The hallmark of a good programmer is the ability to think agnostically about programming languages, focusing first on the problem-solving strategy (the algorithm) before writing the final code. Writing pseudo-code on a whiteboard and discussing it is a critical skill for both interviews and professional work.

**Task 2: Planning First**

## Student Details

**Name:** Aarnav Anoop

**Deakin Student ID:** 223515528

**Tutor: Mr. Xiangwen Yang**

**Class:** Wednesday 6-8pm

**Intended Grade:** Distinction

**Instructions:** Please fill in the module name, along with submission and discussion deadlines from the Ontrack website.

I will adhere to the following timetable for submitting tasks, and will come to class for task discussion with my tutor.

## SIT320 - Time Table

| **Module** | **Task** | **Submission Deadline** | **Discussion Deadline** |
|---|---|---|---|
| Introduction | 1 | 20 Jul 2025 | 27 Jul 2025 |
| Advanced Trees | 2 | 20 Jul 2025 | 27 Jul 2025 |
| Advanced Hashing and Sorting | 3 | 3 Aug 2025 | 10 Aug 2025 |
| Advanced Algorithmic Complexity | 4 | 17 Aug 2025 | 24 Aug 2025 |
| Graphs II | 5 | 17 Aug 2025 | 24 Aug 2025 |
| Dynamic Programming | 6 | 17 Aug 2025 | 24 Aug 2025 |
| Greedy Algorithms | 7 | 24 Aug 2025 | 31 Aug 2025 |
| Linear Programming | 8 | 31 Aug 2025 | 7 Sep 2025 |
| Network Flow Algorithms | 9 | 7 Sep 2025 | 14 Sep 2025 |
| AI Algorithms | 10 | 14 Sep 2025 | 21 Sep 2025 |
| | | | |

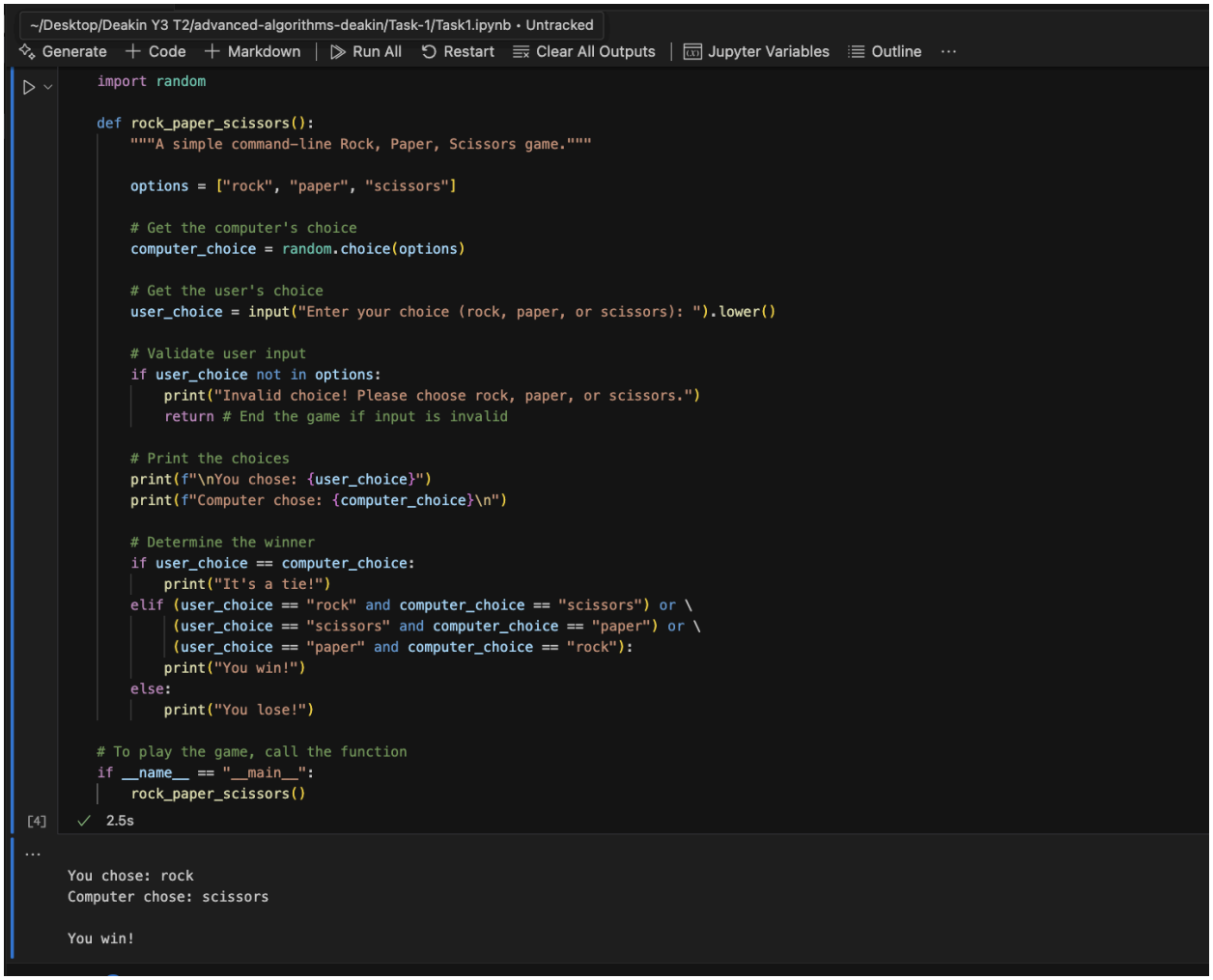
Discussed with your Tutor (Circle one): No
Signatures: Aarnav Anoop

**Task 3: Psuedocode for tic-tac-toe AI:**

```
FUNCTION findBestMove(board):
        bestScore = -infinity
        bestMove = null

        FOR each cell (row, col) on the board:
                IF board[row][col] is empty:
                        place 'X' on board[row][col]
                        moveScore = minimax(board, FALSE)

        remove 'X' from board[row][col]

        IF moveScore > bestScore:
        bestScore = moveScore
        bestMove = the current cell (row, col)

        RETURN bestMove

FUNCTION minimax(board, isMaximizingPlayer):
        finalScore = evaluate(board)
        IF finalScore is 10: RETURN 10
        IF finalScore is -10: RETURN -10
        IF board is full: RETURN 0
```

```
IF isMaximizingPlayer:
        bestScore = -infinity
        FOR each cell (row, col) on the board:
                IF board[row][col] is empty:
                place 'X' on board[row][col]
                score = minimax(board, FALSE)
                remove 'X' from board[row][col]
                bestScore = max(bestScore, score)
        RETURN bestScore


ELSE:
        bestScore = +infinity
        FOR each cell (row, col) on the board:
                IF board[row][col] is empty:
                place 'O' on board[row][col]
                score = minimax(board, TRUE)
                remove 'O' from board[row][col]
                bestScore = min(bestScore, score)
        RETURN bestScore
```

## Task 4: Python Environment Setup:



**Task 5:**
It is a polynomial time algorithm as the size of the table is fixed and does not grow. There is no input size that increases, the algorithm runs at constant time

**Task 6:**
File attached as part of submission

**Task 7:**
A similar approach could be used for games like chess, however upon reflection a few adjustments need to be made.

*The number of trees possible far exceeds computational powers*

The number of possibilities of chess games are far too many for a computer to handle, hence it wouldn't be able to compute this. To overcome this problem we can eliminate the trees in which we deem unnecessary to explore like if one of the nodes already returns an unfavourable outcome then we don't have to explore the other sibling nodes in this tree as it is an assumption that our opponent plays optimally. This is known as alpha beta pruning.

We can also limit the depth at which we look into although this gives us a tradeoff with exploring all possibilities which is something we need to consider.

*The complex heuristics of chess*

While in tic-tac-toe it is easy to measure a game state, it is much more difficult in chess where a more favourable position could be better despite having fewer material points.