Module Assignment & Exam Outlines

Module code and name	DE2-COM2_2019-20 Computing 2	
Module leader	Dr Nicolas Rojas	
Graduate teaching assistants	Angus Clark, Liang He, Icey Lu	
Assessment name & weighting	Individual project: Coursework	100%
Submission date	Week 11: Project code (Closes on Thu 12 Dec 2019 4pm on Blackboard: link 'Project code' in folder *Project*)	

Summary description and objectives for the assessment

TETRILING WITH MISSING PIECES

A polyomino is a plane geometric figure formed by joining one or more equal squares edge to edge. Tetrominoes are polyominoes of four squares; they are popular for their use in the video game Tetris. Tetris pieces can have 19 types of shape, as shown in Figure 1 with an identification number, called *shapeID*, for each shape.

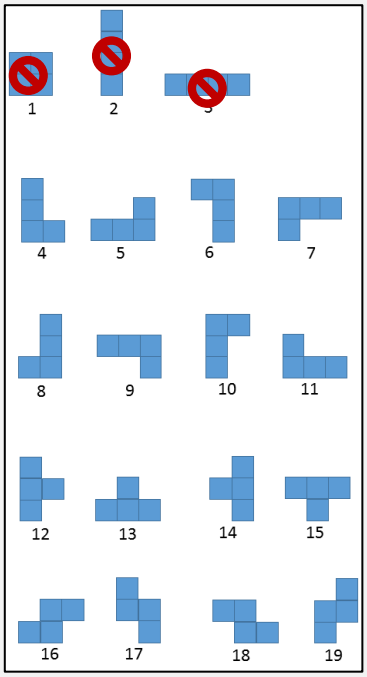

Figure 1. Tetris pieces with shapeID

In this assignment, you will have to design an algorithm to solve the *tetriling with missing pieces* problem. This problem consists of finding a tiling of an arbitrary finite polyomino region by using Tetris pieces, with the exception of the O tetris piece and the I tetris pieces (*shapeIDs* = 1, 2, and 3 in Figure 1)—these are *forbidden pieces*. The objective is to perform the tiling as fast as possible while minimizing the sum of uncovered and added squares in the target region. Figure 2 shows an example of a polyomino region and two possible tilings; this target region is for illustrative purposes only and will not be used in the evaluation of your algorithm.

This individual project will be assessed through your *code* and the results of a *performance test* in which your algorithm (your code) will be tested with three different polyomino regions of increasing complexity and size.

PERFORMANCE TEST

Your proposed algorithm will be evaluated in a performance test, which will assess:

1. The *accuracy* of the solution provided by the algorithm. An accurate solution is one that adjusts well to the given target, minimising the number of excess and missing squares. The value of one excess square is the same as one missing square. A lower score implies a better solution.
2. The *time performance* of the algorithm, measured as the running time that passes until a solution is found. The faster the solution is found, the better.

You are provided with three Python scripts, namely, 'main.py', 'performance_test.py', and 'utils.py', to benchmark your algorithm in similar conditions to those in the performance test. These scripts also provide useful tools such as a random target shape generator, a function to check whether a solution is valid, and a visualisation tool to display a solution against the target shape. *The Python scripts are available on Blackboard (Project > Scripts)*. A simple mock example is provided in the scripts.

It is your responsibility to familiarize with the use of the scripts 'main.py', 'performance_test.py', and 'utils.py', so please start as soon as possible!

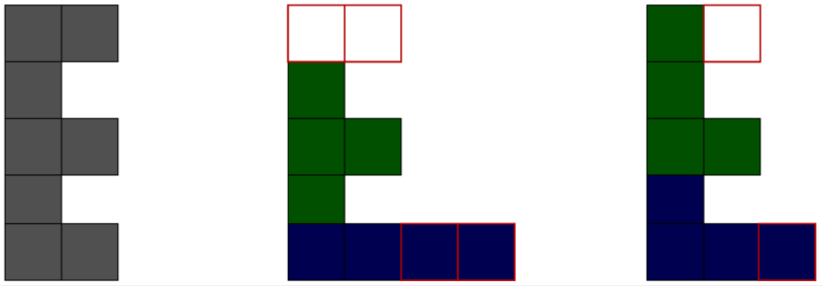

Figure 2. Example of a target region and two possible tiling solutions. ***Left:*** *A target region of 8 squares. This region cannot be exactly tiled using Tetris pieces.* ***Centre****: An approximate tiling using two Tetris pieces whose shapeIDs are 12 and 3. This solution has two uncovered or missing squares at the top (outlined in red) and two excess squares at the bottom (also outlined in red), which add up to a score of 4. Moreover, the solution is invalid as a forbidden piece is used (the I tetris piece with shapeID = 3).* ***Right:*** *A better approximate tiling using two Tetris pieces whose shapeIDs are 4 and 11. This solution has a score of 2 (1 missing square + 1 excess square). The lower score means this is a better solution, and the solution is valid because no forbidden pieces are used.*

OBJECTIVES

The main objective for this assignment is to verify that you are capable of

- Applying algorithm analysis techniques to determine the running time of algorithms and how a programme performs and scales with problem size.
- Explaining the basic principles and foundational techniques for designing algorithmic solutions for unseen problems.
- Using general design methods for devising programmes that are asymptotically efficient.
- Implement well-known, high-level algorithms and adapt them efficiently to new applications.
- Designing, implementing, and analyzing your own algorithms and data structures.

Submission requirements

Failure to comply with any of the requirements below will result in the assignment being marked 0%.

- You are required to submit a ZIP file that includes the source code of your proposed solution in Python 3. The main script of the algorithm should be called 'main.py' which includes the main solution function called 'Tetris' as described in subsection CODE SUBMISSION FORMAT.
- The ZIP file needs to be submitted on Blackboard before Thu 12 Dec 2019 4pm using the link 'Project code' in the folder *Project*.
- It is your responsibility to keep a receipt of the submission (e.g. a screenshot of the Blackboard page after completion of the submission process).

CODE SUBMISSION FORMAT

The main script of your proposed solution should be called 'main.py' which must include a main function called 'Tetris' as shown below:

```
def Tetris(T):
    """
    This function uses the input target matrix, and finally outputs your solution matrix
    :param T: target matrix
    :return: solution matrix
    """
    your code …
    return M
```

This 'Tetris' function takes an arbitrary targeted polyomino region *T* as input, and then generates and returns a tiling solution *M* as output. Specifically, *T* is a matrix provided by the teaching team to represent the target region (the specific shape of the polyomino regions to be used in the

performance test will NOT be disclosed). Each element in matrix *T* represents a square, where '1' means a target square to tile and '0' means a blank square. For example, in Figure 2, the target polyomino region could be represented as:

```
T=[
    [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, 1, 1, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 1, 1, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 1, 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]
]
```

Based on the input target region *T*, the 'Tetris' function will need to generate and return a matrix *M* to represent a tiling solution without using the *forbidden pieces* (pieces whose *shapeIDs* are 1, 2, and 3 in Figure 1). *M* must have exactly the same size (width, height) as *T*. Each element in *M* also represents a square and should be denoted as a tuple: (*shapeID*, *pieceID*), where *shapeID* is an integer which indicates the shape type of the Tetris piece as in Figure 1 and *pieceID* is an integer which univocally identifies which piece does the square belong to, that is, 1 for the first piece, 2 for the second piece, and n for the n^{th} piece. *PieceID* ranges from 1 to n, where n is the total number of Tetris pieces used in the tiling solution, while *shapeID* ranges from 4 to 19 as shown in Figure 1.

For example, the tiling solution in Figure 2 (centre) can be represented as:

```
M=[
    [(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, 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), (0, 0),  (0, 0),  (0, 0), (0, 0), (0, 0)],
    [(0, 0), (0, 0), (0, 0), (12, 1), (0, 0),  (0, 0), (0, 0), (0, 0)],
    [(0, 0), (0, 0), (0, 0), (12, 1), (12, 1), (0, 0), (0, 0), (0, 0)],
    [(0, 0), (0, 0), (0, 0), (12, 1), (0, 0),  (0, 0), (0, 0), (0, 0)],
    [(0, 0), (0, 0), (0, 0), (3, 2),  (3, 2),  (3, 2), (3, 2), (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, 0),  (0, 0), (0, 0), (0, 0)],
    [(0, 0), (0, 0), (0, 0), (0, 0),  (0, 0),  (0, 0), (0, 0), (0, 0)]
]
```

which is invalid as there is a forbidden piece (*shapeID* = 3 in this case). Similarly, the second tiling solution, Figure 2 (right), can be represented as:

```
M=[
    [(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, 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), (4, 2),  (0, 0),  (0, 0),  (0, 0), (0, 0)],
    [(0, 0), (0, 0), (0, 0), (4, 2),  (0, 0),  (0, 0),  (0, 0), (0, 0)],
    [(0, 0), (0, 0), (0, 0), (4, 2),  (4, 2),  (0, 0),  (0, 0), (0, 0)],
    [(0, 0), (0, 0), (0, 0), (11, 1), (0, 0),  (0, 0),  (0, 0), (0, 0)],
    [(0, 0), (0, 0), (0, 0), (11, 1), (11, 1), (11, 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, 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)]
]
```

USE OF EXTERNAL LIBRARIES

Python external libraries (e.g., NumPy, matplotlib) can be used in the proposed solution. However, appropriate comments explaining the use of functions/methods from these libraries have to be included in the code (where their call occurs). Please also include a statement at the top of the 'main.py' file summarising the external libraries and functions/methods used (if any).

PERFORMANCE TEST POLICY

The size of the targets used in the performance test will never exceed a size of 1000x1000, and they can be non-square (e.g., 10x20). It is NOT allowed to use pieces with the shape of shapeIDs = 1, 2, and 3 as in Figure 1; if these types of shapes are identified in your solution, the result will count as invalid. The running time of your algorithm MUST NOT exceed 10 minutes; if the algorithm cannot find a solution within that time in the performance test, the result will count as invalid. All targets used in the performance test will have at least one perfect solution with 0 missing squares and 0 excess squares.

Learning outcomes	**Assessment criteria**
Be able to apply algorithm analysis techniques to determine the running time of algorithms and how a programme performs and scales with problem size.	▪ Appropriate use of principles and paradigms of algorithm design. ▪ Asymptotic efficiency and performance.
Be able to explain the basic principles and foundational techniques for designing algorithmic solutions for unseen problems.	▪ Appropriate use of principles and paradigms of algorithm design. ▪ Quality of comments in code.
Be able to use general design methods for devising programmes that are asymptotically efficient.	▪ Asymptotic efficiency and performance. ▪ Quality of comments in code.
Be able to implement well-known, high-level algorithms and adapt them efficiently to new applications.	▪ Appropriate use of classic, well-known algorithms. ▪ Quality of code readability. ▪ Quality of code structure.
Be able to design, implement, and analyze your own algorithms and data structures.	▪ Elegance and creativity of the proposed algorithmic solution ▪ Quality of code readability. ▪ Quality of code structure. ▪ Quality of comments in code.

Criteria grade descriptors				
Assessment criteria	**≥ 40% (D)**	**≥ 50% (C)**	**≥ 60% (B)**	**≥ 70% (A)**
Asymptotic efficiency and performance. *30%* Please see **Performance test marking system** for the details of mark calculation.	In the performance test, the proposed algorithm provides valid solutions, but it does not adjust well to the given tasks, and the running times are consistently in the slowest group.	In the performance test, the proposed algorithm provides valid solutions, and it starts to adjust well to the given tasks, and the running times are not consistently in the slowest group.	In the performance test, the proposed algorithm provides valid solutions, and it adjust well to the given tasks, and the running times are not in the slowest group.	In the performance test, the proposed algorithm provides valid solutions, and it adjust very well to the given tasks, and the running times tend to be in the fastest group.

Elegance and creativity of the proposed algorithmic solution. *20%*	The proposed algorithm provides valid solutions, but it is cumbersome with unnecessary pieces of code and, or excessively simple but, not effective or efficient.	The proposed algorithm provides valid solutions, and it is effective and efficient but cumbersome and not composable (does not have a structure made of components).	The proposed algorithm provides valid solutions, and it is simple yet effective and efficient, and composable (with a clear structure made of components).	The proposed algorithm provides valid solutions, and it is surprisingly simple and concise yet highly effective and efficient, composable (with a clear structure made of components), and with evidence of out-of-the-box thinking.
Appropriate use of classic, well-known algorithms. *15%*	The proposed algorithm provides valid solutions, but it has no evidence of the use of well-known algorithms discussed in the module or the reading list.	The proposed algorithm provides valid solutions, and some well-known algorithms discussed in the module or the reading list are implemented, but the need of their use is not clear or they are directly unnecessary.	The proposed algorithm provides valid solutions, and some well-known algorithms discussed in the module or the reading list are implemented, but the need of their use is not totally clear.	The proposed algorithm provides valid solutions, and some well-known algorithms discussed in the module or the reading list are implemented, and the need of their use (for all of them) is apparent.
Appropriate use of principles and paradigms of algorithm design. *15%*	The proposed algorithm provides valid solutions, but there is no evidence of the use of principles and paradigms of algorithm design such as reduction, induction, divide and	The proposed algorithm provides valid solutions, and it is mostly based on a greedy algorithm approach without proof of optimality, or based on other principles and	The proposed algorithm provides valid solutions, and there is some evidence of the combined use of principles and paradigms of algorithm design such as reduction, induction,	The proposed algorithm provides valid solutions, and there is clear evidence of the combined use of principles and paradigms of algorithm design such as reduction, induction,

	conquer, greediness, or dynamic programming.	paradigms without a clear justification.	divide and conquer, greediness, or dynamic programming, but the need of their use and applicability is not totally clear.	divide and conquer, greediness, or dynamic programming, and the need of their use and applicability (for all of them) is apparent.
Quality of comments in code. *10%*	The proposed algorithm provides valid solutions, but the code does not have comments explaining the rationale for the algorithm design and implementation choices made.	The proposed algorithm provides valid solutions, and the code have some comments explaining the rationale for some of the algorithm design and implementation choices made. The comments tend to clutter the code.	The proposed algorithm provides valid solutions, and the code have comments explaining the rationale for the majority of the algorithm design and implementation choices made. The comments tend to clutter the code.	The proposed algorithm provides valid solutions, and the code have concise comments explaining the rationale for every algorithm design and implementation choice made. The comments do not clutter the code.
Quality of code readability. *5%*	The proposed algorithm provides valid solutions, but the code does not follow standards of indentation and formatting, variable names do not communicate intent, functions do more than one thing, and nested if statements are ubiquitous.	The proposed algorithm provides valid solutions, and the code follows standards of indentation and formatting, but variable names do not communicate intent, functions do more than one thing, and nested if statements are frequent.	The proposed algorithm provides valid solutions, the code follows standards of indentation and formatting, variable names communicate intent, and the majority of the functions do not do more than one thing. Some guard clauses to avoid	The proposed algorithm provides valid solutions, the code follows standards of indentation and formatting, variable names communicate intent, functions do not do more than one thing, and guard clauses to avoid nested if statements

			nested if statements are used.	are always used.
Quality of code structure. *5%*	The proposed algorithm provides valid solutions, but there is no evidence of isolation of code parts via the use of functions or object-oriented programming.	The proposed algorithm provides valid solutions, and there is some evidence of isolation of code parts via the use of functions.	The proposed algorithm provides valid solutions, and there is clear evidence of isolation of code parts via the use of functions.	The proposed algorithm provides valid solutions, and there is clear evidence of isolation of code parts via the use of functions and object-oriented programming.

Performance test marking system

The proposed algorithm by each student will be evaluated in a performance test, which involves three different polyomino regions of increasing difficulty and size. For each one of these three tasks, marks will be given according to accuracy and time performance.

ALGORITHM ACCURACY

For each one of the three tasks, accuracy will be marked as follows:

$$Acc_i(\%) = \begin{cases} 100 \times \left(1 - \frac{m_i + e_i}{T_i}\right), & m_i + e_i \leq T_i \\ 0, & m_i + e_i > T_i \end{cases},$$

where m_i, e_i and T_i represent the number of missing squares, the number of excess squares, and the total number of squares in the target shape, respectively. The index i indicates the task number (1, 2 or 3). A perfect solution has neither missing or excess squares, granting a 100% mark. The overall algorithm accuracy mark is obtained as follows:

$$Acc_{total}(\%) = \frac{0.5\ Weight_1\ Acc_1 + 0.2\ Weight_2\ Acc_2 + 0.3\ Weight_3\ Acc_3}{2}.$$

Where the weight i is calculated below:

$$Weight_i = \begin{cases} 1 + \left(\frac{Acc_i - 60}{40}\right), & Acc_i \geq 60\% \\ 1, & Acc_i < 60\% \end{cases}.$$

Students whose algorithm cannot find any solution, or will not run due to errors, will get a mark of 0% for this part.

ALGORITHM TIME PERFORMANCE

The running time will be measured for each one of the three target shapes, for every student in the class. If the running time exceeds 10 minutes and a solution has not been found, the test will be stopped and the mark for that task will be 0%.

Three rankings (one for each task) will be made, ordering all the students' results from the fastest to the slowest. The lists will be divided using quartiles, and each student will get their mark depending on the group in which they fall as follows:

Group	**Mark ($Time_i$)**
G_1 (fastest 25%)	100%

G_2	80%
G_3	60%
G_4 (slowest 25%)	40%

$Time_i$ indicates the time performance mark for task i (1, 2 or 3). The overall time performance mark is obtained as follows:

$$Time_{total}(\%) = 0.2\ Time_1 + 0.3\ Time_2 + 0.5\ Time_3.$$

Students whose algorithm cannot find any solution will get a mark of 0% for this part.

ASYMPTOTIC EFFICIENCY AND PERFORMANCE MARK

The equation for calculating the overall algorithm asymptotic efficiency and performance mark, including accuracy and time, is as follows:

$$Asymptotic\ efficiency\ and\ performance\ mark = 0.9\ Acc_{total} + 0.1\ Time_{total}.$$