Module Code
COM00015C

BEng, BSc, MEng and MMath Degree Examinations 2023–24

Department Computer Science

Title Software 1

TIME ALLOWED FIVE hours (Recommended time to complete FOUR hours)

Papers late by up to 30 minutes will be subject to a 5 mark penalty; papers later than 30 minutes will receive 0 marks.

The time allowed includes the time to download the paper and to upload the answers.

Word Limit Not Applicable

Allocation of Marks: Question 1. 3 and 4 are worth 10% each, question 2 is worth 15% and question 5 is worth 35%. The remaining 20% are allocated to style, clarity, code documentation and quality of code.

Instructions:

Candidates must answer **all** questions using Python 3.10 or above. Failing to do so may result in a mark of 0%. All questions are independent and can be answered in any order.

Download the paper and the required source files from the VLE, in the "Assessment>SOF1 2023-24 January exam" section. Once downloaded, unzip the file. You **must** save all your code in the files provided. **Do not** save your code anywhere else other than this folder. Submit your answers to the GradeScope submission point named **SOF1 2023-24 January exam**. You can find the GradeScope submission point on the "Assessment" page on the VLE.

If you have urgent queries regarding a suspected error in the exam, inform `cs-exams@york.ac.uk` with enough time for a response to be considered and made within the first hour of the start of the exam. Corrections or clarifications will NOT be announced after the first hour of the exam. If a question is unclear, answer the question as best you can, and note the assumptions you have made to allow you to proceed. Inform ⟨`cs-exams@york.ac.uk`⟩ about any suspected errors on the paper immediately **after** you submit.

Note on Academic Integrity

We are treating this online examination as a time-limited open assessment, and you are therefore permitted to refer to written and online materials to aid you in your answers. However, you must ensure that the work you submit is entirely your own, and for the whole time the assessment is live you must not:

- communicate with other students on the topic of this assessment.
- communicate with departmental staff on the topic of the assessment (other than to highlight an error or issue with the assessment which needs amendment or clarification).
- seek assistance with the assessment from academic support services, such as the Writing and Language Skills Centre or Maths Skills Centre, or from Disability Services (unless you have been recommended an exam support worker in a Student Support Plan).
- seek advice or contribution from any other third party, including proofreaders, friends, or family members.

We expect, and trust, that all our students will seek to maintain the integrity of the assessment, and of their award, through ensuring that these instructions are strictly followed. Where evidence of academic misconduct is evident this will be addressed in line with the Academic Misconduct Policy and if proven be penalised in line with the appropriate penalty table. Given the nature of these assessments, any collusion identified will normally be treated as cheating/breach of assessment regulations and penalised using the appropriate penalty table (see AM3.3. of the Guide to Assessment).

1 (10 marks) Basic Programming Structure

The code must be written in the provided file `question_1.py`.

Implement a function `string_pattern(size)` that returns a string representing a `X` when printed on the console using `+` and `-` symbols. The size of the `X` is given by the parameter `size`. The function must raise a `ValueError` if `size` ≤ 2.

for example:

```
>>> string_pattern(3)
'+-+\n-+-\n+-+\n'
>>> string_pattern(4)
'+--+\n-++-\n-++-\n+--+\n'
>>> print(string_pattern(4))
+--+
-++-
-++-
+--+

>>> print(string_pattern(5))
+---+
-+-+-
--+--
-+-+-
+---+

>>>
```


Continued.

2 (15 marks) Basic Programming Structure

The code must be written in the provided file `question_2.py`.

In binary signal processing, noise reduction can be achieved by shrinking and expanding elements of the foreground using a structuring element. A binary signal is composed of 1s and 0s, where 1s represent the foreground or elements of interest and 0s are the background. A one-dimensional binary signal is represented as a list of 0s and 1s. A structuring element is also represented by a smaller list of 0s and 1s.

The "shrinking" operation shrinks the boundaries of the foreground regions (represented by 1s). It works by moving a structuring element over the input signal. If all the elements of the structuring element overlap with the signal's 1s, the corresponding position in the output is set to 1; otherwise, it is set to 0 (see Figure 1). This operation is useful for removing noise (object smaller than the structuring element).

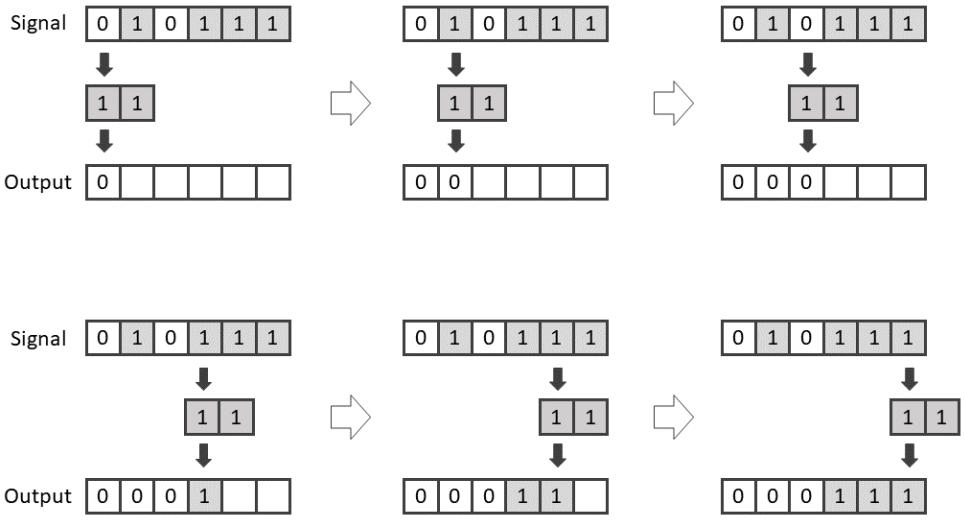

Figure 1: An example of a shrinking operation.

The "expanding" operation is the opposite of shrinking. It is used to expand the boundaries of foreground regions. The operation also involves a structuring element, but it sets the output to 1 for the region overlapped by the structuring element if there's at least one 1 in the overlapping region between the structuring element and the input signal as shown in Figure 2.

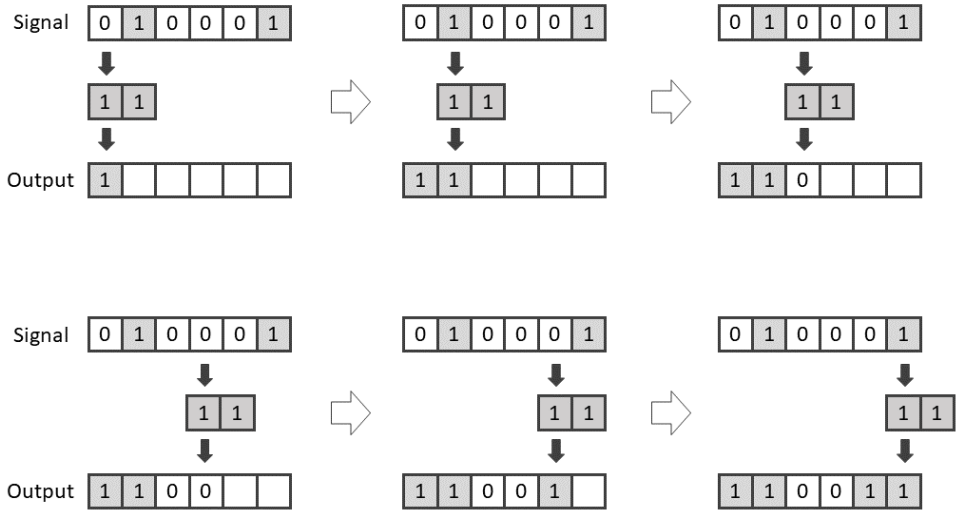

Figure 2: An example of an expanding operation.

(i) [5 marks] Implement a function `shrink(signal, element)` that shrinks the foreground of a binary one-dimensional signal using the structuring element `element`.

(ii) [5 marks] Implement a function `expand(signal, element)` that expands the foreground of a binary one-dimensional signal using the structuring element `element`.

(iii) [5 marks] Implement a function `denoise(signal, element)` that first shrinks and then expands the foreground of a binary one-dimensional signal using the structuring element `element`.

For example:

```
>>> signal = [1, 0, 0, 1, 0, 0, 1, 1, 0, 1]
>>> structuring_element = [1, 1]
>>> shrink(signal, structuring_element)
[0, 0, 0, 0, 0, 0, 1, 0, 0, 1]
>>> expand(signal, structuring_element)
[1, 0, 1, 1, 0, 1, 1, 1, 1, 1]
>>> denoise(signal, structuring_element)
[0, 0, 0, 0, 0, 1, 1, 0, 1, 1]
>>>
```

3 (10 marks) Python's Built-in Data Structures

The code must be written in the provided file `question_3.py`.

The aim of the question is to compute the score of a trick in a game of card. The game use a standard deck of 32 cards.In a standard deck of 32 playing cards, there are four suits: Spades, Hearts, Diamonds, and Clubs. Each suit has eight ranks: the numbers 7 through ten, the Jack, the Queen, the King, and the Ace.

In the game, cards have different rankings and values depending on whether they are in the trump suit or not.

- Trump Suit Ranking and Values are Jack = 20 points, 9 = 14 points, Ace = 11 points, 10 = 10 points, King = 4 points, Queen = 3 points, 8 = 0 points, and 7 = 0 points.
- Non-Trump Suit Ranking and Values are Ace = 11 points, 10 = 10 points, King = 4 points, Queen = 3 points, Jack = 2 points, 9 = 0 points, 8 = 0 points, and 7 = 0 points.

For our problem, a card is represented by a `tuple` of the form `(rank, suit)` where:

- `rank` is one of the following string `'7'`,`'8'`,`'9'`,`'10'`,`'Jack'`,`'Queen'`, `'King'`, and `'Ace'`.
- `suit` is one of the following string `'Spades '`,`'Diamonds'`,`'Hearts'`, and `'Clubs'`.

A trick is a `set` containing exactly four distinct cards.

Implement a function `trick_score(trick, trump_suit)` that returns the score of a trick using the values of each card, given a trump suit. The function must raise a `TypeError` if the `trump_suit` is not a valid suit, and the function must raise a `ValueError` if the trick is not composed of exactly four valid cards.

For example:

```
>>> trick = {('9', 'Clubs'), ('7', 'Hearts'),
             ('Queen', 'Hearts'), ('Jack', 'Hearts')}
>>> print(trick_score(trick, 'Clubs'))
19
>>> trick = {('Jack', 'Clubs'), ('8', 'Hearts'),
             ('King', 'Hearts'), ('9', 'Hearts')}
>>> print(trick_score(trick, 'Clubs'))
24
>>>
```

4 (10 marks) Recursion

The code must be written in the provided file `question_4.py`.

A **palindromic number** is a number that remains the same when its digits are reversed. For example, 0, 1, 2, 33, 44, 101, 111, and 121 are all palindromic numbers. Note that a number cannot start with one or more $0s$ with the exception of the number 0. For example, 010 and 001100 are not numbers and therefore are not palindromic numbers. The aim of this question is to find the set of the longest palindromic numbers (that is having the most digits) within a given number.

For example:

- given the number 1991, the set of longest palindromic numbers contains only one element $\{1991\}$ as 1991 is a palindromic number,
- given the number 199132002, the set of longest palindromic numbers is $\{1991, 2002\}$,
- finally, given the number 13254, the set of longest palindromic numbers is $\{1, 2, 3, 4, 5\}$.

To solve the problem, consider the following:

1. if the number is palindromic, then it is the (only) longest palindromic number.
2. Otherwise, find the longest palindromic numbers from the number without its first digit,
3. and find the longest palindromic numbers from the number without its last digit.
4. Compare the results from steps 2 and 3, and return the appropriate set of numbers.

Implement a **recursive** function `longest_palindromic_numbers(number)` that returns the `set` of the longest palindromic numbers within the argument `number`. For simplicity, numbers are represented as a string.

Important note: If the implemented solution is not recursive, a mark of 0% will be awarded for this question. You may want to use a recursive helper function.

For example:

```
>>> print(longest_palindromic_numbers('019910'))
{'1991'}
>>> print(longest_palindromic_numbers('199132002'))
{'1991', '2002'}
>>> print(longest_palindromic_numbers('13254'))
{'1', '4', '2', '5', '3'}
>>>
```

5 (35 marks) User Defined Data Structure

The code must be written in the provided file `question_5.py`.

We want to develop a digital version of a French game called "Bazar Bizzare". This game includes 5 wooden pieces shown in Figure 3, and a set of cards where each card has two objects drawn on it. A subset of five cards is shown in Figure 4.

Figure 3: The five wooden pieces of the game: a green bottle, a white ghost, a red sofa, a blue book, and a grey mouse.

The basic rules are: the five wooden pieces are arranged in front of you. Then, a card is returned. If one of the objects is identical, grab the corresponding piece otherwise, catch the only one that has nothing in common with the card: neither shape nor colour. For example, considering the cards shown in Figure 4, the players should grab the blue book for the left most card (identical object), and the grey mouse for the other four cards as no identical objects are present on the card and neither the mouse nor the grey colour are present on the cards.

Figure 4: A subset of the 60 cards contained in the game.

(i) [10 marks] Implement a class `GameObject` representing a single wooden piece. The class should implement the `__init__` method having two string parameters, the first one being the shape of the piece and the second one its colour. The shape of the piece must be stored in the attribute `_shape` and the colour in the attribute `_colour`.

The class must define two properties `shape` and `colour` that allows only read access to the attributes `_shape` and `_colour`, but does not allow modification of the attributes.

Finally, the class must overload the operator `==`. Two `GameObject` instances are equal if and only if they have the same shape and colour.

(ii) [10 marks] Implement the class `GameCard` representing a single card. The class should implement the `__init__` method having two `GameObject` parameters corresponding to the objects drawn on the card. A **copy** of the two objects should be stored in the instance attribute `_content`, which is a `list` of `GameObject`.

The class must define the property `content` that allows only read access to the attributes `_content` but does not allow modification of the attributes. In addition, the returned value of the property must be a **deep copy** of the attribute `_content`.

Finally, the class must overload the operator `==`. Two `GameCard` instances are equal if and only if they contain the same objects. The order of the objects does not matter.

For example:

```
>>> card1 = GameCard(GameObject('bottle', 'green'),
                     GameObject('book', 'red'))
>>> card2 = GameCard(GameObject('book', 'red'),
                     GameObject('bottle', 'green'))
>>> card1 == card2
True
>>> card3 = GameCard(GameObject('book', 'green'),
                     GameObject('bottle', 'red'))
>>> card1 == card3
False
>>>
```


Continued.

(iii) [15 marks] Implement the class `CardDeck`

(a) [5 marks] Implement the `__init__` method. The method takes only one parameter which is a `list` of `GameObject` representing the wooden pieces of the game. The method must raise a `ValueError` if the list does not contain three, four, or five objects. In addition, the method must also raise a `ValueError` if two objects in the list have the same shape or colour.

(b) [10 marks] Implement the method `generate_deck` that does not take any parameter, and returns the list of all possible valid cards for the game. The type of the returned value is a `list` of `GameCard`. A card is valid if and only if it allows to pick exactly one wooden piece. For example, given the set of five wooden pieces shown in Figure 3, a card containing a blue book and a red chair is not valid as we could pick the blue book and the red chair. Similarly, a card containing a blue chair and red book is not valid, as you could pick the grey mouse, the white ghost and the green bottle.

For example, the method `generate_deck()` for an instance of `CardDeck` initialised using only three wooden pieces would return a list of nine cards. An example of all possible cards for a game consisting of three wooden pieces (green bottle, red chair, blue book) is shown in Table 1. When using 4 wooden pieces, 48 cards are possible and when using 5 wooden pieces, 120 cards are possible.

Table 1: The 9 possible cards when using only 3 wooden pieces: a green bottle, a red chair and a blue book.

(<chair, green>, <bottle, red>)	(<chair, green>, <book, blue>)
(<chair, blue>, <bottle, green>)	(<chair, blue>, <book, red>)
(<chair, red>, <bottle, blue>)	(<chair, red>, <book, green>)
(<bottle, green>, <book, red>)	(<bottle, blue>, <book, green>)
(<bottle, red>, <book, blue>)	

End of examination paper