Semester I Examinations 2021/2022
Solutions/Grading Scheme

Exam Code(s)	1MAI1, 1CSD1, 1SPE1, 2SPE1
Exam(s)	MSc in Computer Science (Artificial Intelligence), MSc in Computer Science (Artificial Intelligence) - Online
Module Code(s)	CT5132, CT5148
Module(s)	**Programming and Tools for Artificial Intelligence**
Discipline	School of Computer Science
Paper No.	1
External Examiner	Dr. John Woodward
Internal Examiner(s)	Prof. Michael Madden
	Dr. James McDermott *
Programme Coordinator(s)	Dr. Matthias Nickles, Dr. James McDermott
Instructions	**Answer all 4 questions. All are worth equal marks.**
Duration	2 Hours exam
Number of pages	**??** (including this page)
Discipline	Computer Science
Requirements	
Release in Exam Venue	Yes ☒ No ☐
Release to Library	Yes ☒ No ☐
Calculator	Allowed (non-programmable)

Question 1: Basic Python

(a) State the error in this function which is intended to calculate the factorial of input `n`. **[5]**

```
def factorial(n):
    '''return n!, assuming n is integer'''
    if n <= 0:
        print(1)
    else:
        print(n * factorial(n - 1))
```

Solution. It prints, instead of returning as stated [3]. The recursive case will fail as

```
factorial(n-1)
```

has value None. [2] **Common errors**. The case where n = 0 is not a problem.

(b) In the game of tic-tac-toe (also known as noughts and crosses), two players place an `X` or an `O` respectively in cells in a 3x3 grid. Name a Python data structure suitable for storing the grid, and write a snippet of code which would initialise an empty grid and then place an `X` in the bottom-left corner. Use pure Python, no imported libraries. **[5]**

Solution. A list of 3 lists, each of length 3 [2]. Code below [3].

```
grid = [
  [None, None, None],
  [None, None, None],
  [None, None, None]
]
grid[2][0] = 'X'
```

(c) The following code is correct. Explain what `int` does here and why this usage of `int` is correct, including why we do not write `0`, `int()` or `int(0)`. **[5]**

```
# count occurences of unique items in list L, and
# store results in dictionary d.
d = defaultdict(int)
for x in L:
   d[x] += 1
```

Solution. Here `int` is supplied as the initializer for the value for new key-value pairs in defaultdict [2]. defaultdict expects a function (or another callable), and it will call that function. If we wrote 0 or int() or int(0) (all are the same thing), it would try to call 0. int() returns 0, so supplying int means it will call a function which returns 0.

(d) Given the following data structure: **[5]**

```
T = ('a', ('b', 'c'), ('d', ('e', 'f', ('g', 'h'), ('i', 'j'))))
```

Write an expression using `T` that will have the value `('i', 'j')`.

Solution `T[2][1][3]`. **Grading**. 2 for something involving T and square brackets. 5 for completely correct.

(e) Why is `list` needed here? Why is `map` designed this way? **[5]**

```
L = ['ambulance', 'anteater', 'aardvark']
print(list(map(len, L)))
```

Solution. `map()` returns an iterator, not a list. [The iterator does not contain any results of applying the function `len`.] Trying to print that iterator will print something like `<map at 0x12341234>`. Calling list() forms a concrete list which can be printed. [3] This allows us to construct code that processes large sequences of data (larger than memory, even "infinite") without bringing the whole sequence into memory at once. [2]

Question 2: Advanced Python

(a) In Python, what are the differences between `math.nan` and `math.inf`? Illustrate with examples of how they arise in numerical code and how they behave in boolean operators such as `>`. **[5]**

Solution. `nan` is "not a number", e.g. 0/0, while `inf` is positive infinity, e.g. 1/0. `math.nan > 3` is False, but `math.inf > 3` is True. **Grading** 1 each for inf and nan creation, and 3 for clear differences. **Common error**. A lot of people wrote that nan indicates that the value is a non-numerical type, eg is a string. That's not correct – nan is a float, just a special float. If you try eg `float('hello')`, you will get a ValueError, not a nan. BTW, some people were ridiculing Python's approach here, but in fact the inf and nan values and their interaction with arithmetic are part of IEEE754, the floating-point standard which all modern languages use. A lot of students wrote that nan "returns" something, but nan is a float, not a function.

(b) Briefly describe the conceptual implementation of the *dictionary* (also known as *hash table*). Refer to table size, `hash`, and the modulus operator `%`. State and explain the time complexity of writing and retrieving items to/from the dictionary. You can ignore the issue of hash collisions. **[10]**

Solution. Conceptually, a hash table is a list L of m entries, each (None, None). We choose m arbitrarily, usually a small integer eg 100. To store a key-value pair (k, v), we calculate the slot number `i = hash(k) % m`. We write `L[i] = (k, v)`. To retrieve the key k, we use the same calculation, and retrieve the element L[i]. If it is (None, None), then the key k is not present. If it is not None, let's say it is (k2, v). If k == k2, then we retrieve the value v. All of the operations are O(1) with respect to n the number of elements added to the dictionary so far.

Common errors. Failing to specify that the implementation is like a list or array. Confusing the hash and the modulus. Failing to mention that we actually store the (key, value) pair (not just the value) at location i. Guessing O(n) – remember, the O(1) performance is one of the minor miracles of computer science (– Downey)!

(c) What are the properties of the factorial function which make it suitable for memoisation? **[5]**

Solution. Deterministic, no side-effects [2]. Function is called with same arguments many times [2]. Can be long-running. [1]

Common errors. A lot of students think memoisation applies only to recursive functions, which is not correct.

(d) What is special about `deque`, the double-ended queue object available in Python `collections`? Explain where this is useful. **[5]**

Solution It is implemented to allow fast O(1) access and fast deletion at both ends. This is useful if we need to manage a queue where items are continually arriving (joining the end of the queue, append()) and being processed (being removed from the other end, popleft()). This could happen eg in a HPC batch-job queueing system, or any first-in, first-out queue. **Grading**. 1 for "both ends", 1 for O(1), 1 for deletion at left. 2 for example. **Common errors**. Statements such as "deque allows removal at both ends" – but notice we can remove an item from either end of a list quite easily, the point is that for a list, the operation is slow. Many students did not give an example.

Question 3: Data Science

(a) The following code fragment uses loops. Rewrite the fragment in a Numpy vectorised style. Assume `L` is a list of `float` values. **[5]**

```
s = 0
for i in range(len(L) - 1):
    if L[i] < L[i+1]:
        s += 1
```

Solution. `x = np.array(L); s = (np.diff(x) > ).sum()`. Another good solution: `x = np.array(L); s = (x[:-1] < x[1:]).sum()`.

Grading 1 for x, 2 for a vector operation +1 if it's correct, +1 for sum. **Common errors**. Writing this as a list comprehension, or a for-loop over a Numpy array, is not vectorisation – it won't be fast because the loop is still in Python, not in the underlying library. Writing `x > x` obviously won't work, and anything involving an index such as `x[i]` is not vectorised.

(b) Explain the concept of *batch job* systems in high-performance computing, including advantages and disadvantages. **[5]**

Solution. A batch job is a large/long-running compute job which is submitted by a user to a HPC machine (a supercomputer/cluster/whatever), for later processing, without user interaction. It is held in a queue until the appropriate time to run it, and the user is notified when finished. [2] The advantage is that the HPC system can schedule jobs in an efficient way, eg starting one job immediately when another finishes, to avoid idle time. [2] The disadvantage is that the user has to wait for their results. This is particularly difficult when there are possible errors as the debug cycle is lengthened. [1] **Common errors**. Confusing a batch job system with the concept of batches in machine learning training. Omitting the point that there is a queue, or no interaction.

(c) Rewrite the following R code using the `magrittr` pipe, where `D` is a tibble and `a` and `b` are columns. **[5]**

```
select(mutate(D, a=0.4*a, b=0.4*b), year, a, b)
```

Solution.

```
D %>% mutate(a=0.4*a, b=0.4*b) %>% select(year, a, b)
```

Grading. 2 for any kind of unwind/rewrite, 1 for still using mutate and select, 2 for fully correct.

(d) The following data is not *tidy*. Explain why not, and show what it would look like in tidy format. **[5]**

Country	Metric	2019	2020
Ireland	Population	5.1	5.2
	GDP	101	102
France	Population	71	72
	GDP	400	410

Solution. It's not tidy, because each column is not a variable, eg the 2019 column contains values from multiple variables.

Country	Year	Population	GDP
Ireland	2019	5.1	101
Ireland	2020	5.2	102
France	2019	71	400
France	2020	72	410

(e) Suppose we have two dplyr tibbles named `rentals` and `customers`, as shown below. Notice that not every customer ID has an entry in the `customers` table. Write a dplyr join to create a tibble containing all rentals together with the corresponding names and addresses. Names and addresses should be blank wherever they are not available. **[5]**

Rentals table

Date	Movie ID	Customer ID
01-Jan	102	1
02-Jan	101	2
02-Jan	102	3
05-Jan	103	1
05-Jan	104	7

Customer table

Customer ID	Name	Address
1	Bob	11, Haight St
2	Frida	Oxford Circus
3	Carrie	99, Fifth Ave

Solution.

```
left_join(rentals, customers, by="CustomerID")
```

. A full join was also ok. A right join was not. SQL code was not accepted.

Question 4: Tools and Applications

The following programs illustrate the legal syntax in a simple programming language called ACIDIC.

```
10 PRINT 1
20 GOTO 10
```

```
1 PRINT 2
2 SET X[0] 5
3 SET X[1] (X[0]+5)
4 GOTO X[0]
```

An ACIDIC program consists of 1 or more lines. Every line consists of a line number and a command, separated by a space. There are several commands: `PRINT`, `SET` (for variable assignment), `IF`, `GOTO` (jump to the given line number).

Every command is followed by one argument (again separated by a space). The argument can involve numerical constants, variables, and operators (e.g. `1`, `10`, `X[0]`, `X[9]`, `+`, `*`, `>`, `==`). The result of executing a boolean operator such as `==` is an integer 0 or 1. The `SET` command is an exception as it takes *two* arguments: the variable name, and the value to assign to that variable.

Variables `X[0]` up to `X[9]` are the only ones allowed. The `IF` command takes one argument, an expression. If its value True or is a number greater than 0, the next line is executed, otherwise it is not executed. Finally, line numbers need not be successive in the code, as the ACIDIC interpreter will sort them before execution.

(a) Write out a grammar to generate programs in this language. Include all of the commands, constants, variables, and operators mentioned above. Notice that Example 2 above is legal, but will crash, as line 4 tries to `GOTO X[0]`, i.e. to line number 5, which does not exist. Your grammar only has to generate legal programs, and does not have to prevent programs which would crash. **[15]**

(b) Consider the following ACIDIC program. Draw a *graph* representing the flow of the program, where nodes are line numbers. Also, write the adjacency matrix for this graph. **[10]**

```
10 GOTO 30
20 GOTO 40
30 GOTO 20
40 GOTO 30
50 PRINT 'Hello'
```

Solution. Here `<program>` is the start rule.

```
<program> ::= <line> | <line>'\n'<program>
<line> ::= <lineno> <cmd>
<lineno> ::= 0 | 1 | 2 | (etc)
<cmd>  ::= <print> | <set> | <if> | <goto>
<print> ::= PRINT <expr>
<set> ::= SET <var> <expr>
<if> ::= IF <expr>
<goto> ::= GOTO <expr>
<expr> ::= <var> | <const> | (<expr> <binop> <expr>)
<var> ::= X[<varidx>]
<varidx> ::= 0 | 1 ... | 9
<const> ::= 0.1 | 1.0 | 10 | etc
<binop> ::= * | + | > | ==
```

The graph should be directed (edges as arrows), showing 10 leading into the 30, 20, 40 cycle. 50 is an isolated node (no arrows) but it's ok to omit it also. [5] The adjacency matrix should be square, binary, and asymmetric. [5]

By convention, we usually write asymmetric adjacency matrices with the "source" nodes listed in the first column and the "destination" nodes listed in the first row. But the other way around is fine too for grading.

	10	20	30	40	50
10	0	0	1	0	0
20	0	0	0	1	0
30	0	1	0	0	0
40	0	0	1	0	0
50	0	0	0	0	0

Grading. 8 marks for any grammar that addresses the question even with multiple flaws. Generous marks up to 12 for flawed solutions. No more than 12 if grammar omits recursion.

Common errors. Many students were able to produce a fairly complete grammar which could produce a single line, but almost no-one was able to implement the recursive rule to produce a program consisting of multiple lines. The crucial point here is not the newline character but the recursion. Recursion is also needed to produce nested expressions such as `(4 * (3 + 2))` A few students wrote things like `<IF> ::= True | False`, but notice that the IF command should be followed by an expression. A command like `IF TRUE` would not be useful.

A few students provided an adjacency list instead of an adjacency matrix, but still received some marks.

A student messaged me with the concern that ACIDIC is an ancient language, but in fact ACIDIC was just invented for this exam. This is because implementing a grammar for a real programming language would be too large for such an exam question.