Part 1: Algorithmic Thinking

Question 1

[10 marks]

Evaluate the following expressions, and provide the output in each case.

(a) `('comp', 10000 + 1)`

A: `('comp', 10001)`

(b) `sorted({'dogs': ['andrew'], 'ducks': ['tim']})[-1]`

A: `'ducks'`

(c) `"c" + str(0)*2 + "lollipop"[:1][0]`

A: `'c00l'`

(d) `bool(range(2, 10, 2) and "dig" in "don't wait")`

A: `False`

(e) `[i for i in "how much wood could a woodchuck chuck".split() if len(i) > 6]`

A: `['woodchuck']`

Question 2

[8 marks]

Rewrite the following function, replacing the `while` loop with a `for` loop, but preserving the remainder of the original code structure:

```
def all_alpha(word):
    i = 0
    while i < len(word):
        if not word[i].isalpha():
            return False
        i += 1
    return True
```

A:

```
def all_alpha(word):
    for i in range(len(word)):
        if not word[i].isalpha():
            return False
    return True
```

Question 3

[10 marks]

The following code is intended to read in a dictionary of words from a file (`dictionary.txt`), spell-check a second file (`jabberwocky.txt`) relative to it, and compute a sorted list of unique words not found in the dictionary.
As presented, the lines of the function are out of order. Put the line numbers in the correct order and introduce appropriate indentation (indent the line numbers to show how the corresponding lines would be indented in your code).

```
1  oov[word] = True
2  word = word.strip(".,;:!?\"\'`")
3  if word.lower() not in dictionary:
4  worddict = {}
5  oov = {}
6  for line in open(doc, encoding="utf-8").readlines():
7  worddict[word.strip()] = True
8  for word in line.split():
9  newwords = out_of_dict("jabberwocky.txt", init_dict("dictionary.txt"))
10 for word in open(dictfile, encoding="utf-8").readlines():
11 return worddict
12 def init_dict(dictfile):
13 return sorted(oov)
14 def out_of_dict(doc, dictionary):
```

A:

```
12
    4
    10
        7
    11
14
    5
    6
        8
            2
            3
                1
    13
9
```

Question 4

[12 marks]

The following function `valid_url(url)` that takes a single string argument `url`, and is intended to return `True` if `url` is a valid full-specified URL (with the hostname ending in `.com` or `.au`), and `False` otherwise. Recall that a valid fully-specified URL is made up of a scheme, hostname, optional port and optional path.

```python
def is_valid_url(url):
    SCHEMES = ("http", "ftp")
    TLDS = ("com", "au")
    try:
        scheme, rest = url.split("://")
    except ValueError:
        return False
    if scheme not in SCHEMES:
        return False
    rest_path = rest.split("/")
    hostname = rest_path[0]
    path = rest_path[1:]
    if ":" in hostname:
        (hostname, port) = hostname.split(":")
        try:
            port = int(port)
        except ValueError:
            return False
    tld = hostname.split(".")[-1]
    if tld not in TLDS:
        return False
    return True
```

The provided implementation has logic errors in it, and rejects some valid fully-specified URLs it should accept, and conversely, accepts some inputs which are not valid fully-specified URLs.

(a) Provide an example of a valid fully-specified URL for which `valid_url()` would return `True` (i.e. the function would perform correctly).

A: *Any URL with the http or ftp scheme, with or without a port number, and with or without a trailing path; satisfied if URL looks plausible (irrespective of whether exists or not)*

(b) Provide an example of a valid fully-specified URL for which `valid_url()` would return `False`, despite it being valid.

A: *Different scheme (e.g. https) OR different TLD; not fussed about whether the hostname exists or not, but the scheme must be valid*

(c) Provide an example of an *in*valid fully-specified URL for which `valid_url()` would return `True`, despite it being invalid.

A: *hostname not fully-specified (e.g.* `http://au`*)*

(d) Provide an example of an *in*valid fully-specified URL for which `valid_url()` would return `False` (i.e. the function would perform correctly).

A: *Lots of possibilities (NB the URL simply needs to be non-fully-specified, so simply omitting the scheme is good enough)*

Question 5

[9 marks]

The following function is meant to recursively sum up the numbers in a (possibly nested) list of numbers, ignoring any non-numeric values, and non-list types. The following is an example function call which illustrates its intended behaviour:

```
>>> sum_list(['b', 1, ['a'], 3, [1, 2, {2: 3}, [[[[4]]]]]])
11
```

Identify exactly three (3) errors in the code (using the provided line numbers), determine for each whether it is a "syntax", "run-time" or "logic" error, and provide a replacement line which corrects the error.

```
1  def sum_list(curr_list):
2      curr_sum = None
3      if type(curr_list) == list:
4          for element in curr_list:
5              element_type = type(element)
6              if element_type = list:
7                  curr_sum += sum_list(element)
8              else:
9                  if element_type == int or float:
10                     curr_sum += element
11     return numsum
```

A:

- *line 2: logic error;* `curr_sum = None` → `curr_sum = 0`
- *line 6: syntax error;* `if element_type = list:` → `if element_type == list:`
- *line 9: logic error;* `if element_type == int or float:` → `if element_type == int or element_type == float:`
- *line 11: run-time error;* `return numsum` → `return curr_sum`

Part 2: Constructing Programs

Question 6

[10 marks]

The function `csv_aggregate(infile, outfile)` takes an input file name `infile` (a string) and output file name `outfile` (also a string). `infile` is of the following format:

```
docid,sentid,pos
0,0,0
0,1,1
1,0,0
1,1,0
1,2,0
```

where each (non-header) row contains (in order) the ID of a document, the ID of a sentence within that document, and a binary value indicating whether the given sentence has positive sentiment (i.e. says positive things) or not. The function takes `infile` and calculates the relative proportion of positive sentiment sentences in a given document, and returns a single row for that document, containing the document ID (`docid`) and the calculated proportion (`pos_ratio`) of positive sentiment sentences. This is saved in the file `outfile`. For example, given the file `sentences.csv` with the content above, `outfile` will be as follows:

```
docid,pos_ratio
0,0.5
1,0.0
```

Provide code to insert into each of the numbered boxes in the code on the next page to complete the function as described. Note that your code will be evaluated at the indentation level indicated for each box.

```
import csv

def csv_aggregate(infile, outfile):
    [        1        ]
    fpout.write("docid,pos_ratio\n")
    prev_docid = None
    [        2        ]
    for row in csv.DictReader(open(infile)):
        [        3        ]
            if prev_docid != None:
                fpout.write("{},{}\n".format(prev_docid,pos/total))
            [        4        ]
            pos = total = 0
        if row["pos"] == '1':
            pos += 1
        total += 1
    if prev_docid != None:
        [        5        ]
    fpout.close()
```

A:

(1)
```
        fpout = open(outfile, "w")
```

(2)
```
        pos = total = 0
```

(3)
```
        if row["docid"] != prev_docid:
```

(4)
```
        prev_docid = row["docid"]
```

(5)
```
        fpout.write("{},{}\n".format(prev_docid,pos/total))
```

Question 7

[12 marks]

Write a function `equiword(word)` that takes a single argument `word` (a non-empty string) and returns a (positive) integer `n` if all unique letters in `word` occur exactly `n` times, and `False` otherwise. For example:

```
>>> equiword("intestines")
2
>>> equiword("deeded")
3
>>> equiword("duck")
1
>>> equiword("doggy")
False
```

A:

```
def equiword(word):
    from collections import defaultdict
    letter_dict = defaultdict(int)
    for letter in word:
        letter_dict[letter] += 1
    letter_freq_list = sorted(letter_dict.values())
    if word and letter_freq_list == letter_freq_list[::-1]:
        return letter_freq_list[0]
    return False
```

Question 8

[18 marks]

A "Takuzu" puzzle takes the form of an $n \times n$ square of even dimensions (e.g. 4×4 but not 5×5), each cell of which is to be filled with either a zero (`0`) or a one (`1`), such that, when solved:

- each row and column of the square contains an equal number of zeroes and ones; and
- no row or column contains more than 2 adjacent occurrences of the same digit.

For example, the following is a valid (completed) Takuzu square:

0	1	0	1
1	0	1	0
1	1	0	0
0	0	1	1

as is the following a valid (partially-solved) Takuzu square:

		0	1
1	0		
1		0	
	0		

whereas the following is not, because the first row and third column contain too many ones:

0	1	1	1
1		1	0
1	1	0	
0	0	1	1

and the following is similarly invalid, because it contains three adjacent ones in the first and fourth columns, and three adjacent zeroes in the third row:

0	1	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	1
1	0	0	1	1	0
0	0	1	1	0	1
0	1	0	1	0	1

We will represent Takuzu squares based on the following two-dimensional list representation (corresponding to the valid 4×4 solved example from the previous page), noting that the number of rows and columns will vary according to the size of the square:

```
takuzu = [[0, 1, 0, 1],
          [1, 0, 1, 0],
          [1, 1, 0, 0],
          [0, 0, 1, 1]]
```

For partially-solved Takuzu squares such as the following (which is valid):

		0	1
1	0		
1		0	
	0		

blank cells are represented by the value `None`, as follows:

```
takuzu = [[None, None, 0, 1],
          [1, 0, None, None],
          [1, None, 0, None],
          [None, 0, None, None]]
```

That is, there are three possible values for each cell: `None`, `0` and `1`. Note that, for partially-solved Takuzu squares, the number of zeroes and ones in a given row or column may not be balanced, because of the presence of blank cells. Note also that, for the purposes of this question, an invalid square is one that violates one of the two constraints described above, and we are not concerned with the question of whether it has a unique solution or not, or the degree to which it is partially solved. As such, the following is a valid (partially-solved) square:

```
takuzu = [[None, None, None, None],
          [None, None, None, None],
          [None, None, None, None],
          [None, None, None, None]]
```

You may assume that all Takuzu inputs provided as arguments to your function are square, of even dimensions, at least of size 2×2, contain one of the three possible values in each cell, and are either solved or partially solved.

Write a function `valid(takuzu)` that takes a Takuzu square as an argument and returns `True` if `takuzu` is a valid Takuzu square, and `False` otherwise.
For example:

```
>>> valid([[None, None, 0, 1], [1, 0, None, None], [1, None, 0, None],
... [None, 0, None, 1]])
True
>>> valid([[None, None, 0, 1], [1, 0, None, None], [1, None, 0, None],
... [1, 0, None, 1]])
False
>>> valid([[1, 0, 0, 1], [0, 0, 1, 1], [1, 1, 0, 0], [1, 0, 1, 0]])
False
>>> valid([[1, 0], [0, 1]])
True
>>> valid([[None, None], [None, None]])
True
>>> valid([[1, 1], [0, 1]])
False
```

A:

```
def valid(takuzu):
    MAX_VALS = len(takuzu)/2

    def valid_rowcol(lst):
        MAX_REPEATS = 2
        counts = [0, 0]
        prev = None
        repeats = 0
        for cell in lst:
            for val in range(len(counts)):
                if cell == val:
                    counts[val] += 1
                    if prev == val:
                        repeats += 1
                        if repeats > MAX_REPEATS:
                            return False
                    else:
                        repeats = 1
            if cell is None:
                prev = None
                repeats = 0
            prev = cell
        if counts[0] > MAX_VALS or counts[1] > MAX_VALS:
            return False
        return True

    for row in takuzu:
        if not valid_rowcol(row):
            return False
    for colno in range(0, len(takuzu)):
        if not valid_rowcol([takuzu[i][colno] for i in range(0, len(takuzu))]):
            return False
    return True
```

Part 3: Conceptual Questions

Question 9: Algorithmic Problem Solving

[9 marks]

(a) In the context of algorithmic development, what is the difference between an "exact" and "approximate" approach?

A: *An exact approach provides an exact solution, whereas an approximate approach provides an estimate of the solution (which may or may not be the exact value)*

(b) Outline the "divide-and-conquer" strategy of algorithmic problem solving, with the aid of an example.

A: *Solve a problem by solving its sub-problems ...*

Question 10: Applications of Computing

[10 marks]

(a) With the aid of an example, describe what "alignment" means in the context of statistical machine translation, e.g. as used for language decipherment purposes.

A: *Alignment is the process of determining which substrings in the target language are translations of which particular substrings in the source language*

(b) List three (3) possible attacks on an "Internet voting system".

A: *Possible answers:*

- *privacy breach on the client machines*
- *privacy breach in transit*
- *privacy breach on the server*
- *coercion based on physical presence*
- *coercion based on electronic privacy invasion*
- *impersonation (i.e. authentication failures)*
- *multiple voting*
- *Denial-of-service attacks on the server*

Question 11: HTML and the Internet

[12 marks]

(a) Based on the following HTML document:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>HTML Example</title>
5  </head>
6  <body>
7  <ul>
8  <li>A <a href="http://en.wikipedia.org/wiki/Link">link</a>,</li>
9  <li>a link;</li>
10 <li>my kingdom for a link!</li>
11 </ul>
12 </body>
13 </html>
```

and the provided line numbers, identify in the document where each of the following items occurs. In the case that the item spans multiple lines, you should specify the full range of line numbers (e.g. `2-4`).

[8 marks]

(i) A document type declaration:

A: *line 1*

(ii) A tag attribute:

A: *line 8*

(iii) Anchor text:

A: *8*

(iv) A list:

A: *lines 7–11 OR 8–10*

(b) What is the relationship between a "hostname" and an "IP address"?

A: *hostnames are hierarchically-organised, human-readable versions of machine addresses map onto IP addresses by DNS*