2019 SH2 Preliminary Examination – Practical – Suggested Solutions and Marking Scheme

Question 1

Task 1.1 Solution - Evidence 1

```
fileHandle18 = open("WEATHER-JUNE-2018.TXT")
data18 = []
for line in fileHandle18:
    data18.append(line.strip().split("\t"))
fileHandle18.close()

fileHandle19 = open("WEATHER-JUNE-2019.TXT")
data19 = []
for line in fileHandle19:
    data19.append(line.strip().split("\t"))
fileHandle19.close()

for i in range(len(data18)):
    data18[i] = ["2018"] + data18[i]

for i in range(len(data19)):
    data19[i] = ["2019"] + data19[i]
```

Task 1.1 - Evidence 1 Mark Allocation

- All data from both files read and stored [1]
- Insertion of the year value to differentiate entries from both files (may occur in a later part) [1]

Task 1.2 Solution – Evidence 2

```
def bubbleSort(data):
    for i in range(len(data) - 1):
        didSwap = False
        for j in range(len(data) - i - 1):
            if float(data[j][6]) > float(data[j + 1][6]):
                data[j], data[j + 1] = data[j + 1], data[j]
                didSwap = True
        if not didSwap:
            break
    return data

sortedData18 = bubbleSort(data18)
sortedData19 = bubbleSort(data19)
```

Task 1.2 – Evidence 2 Mark Allocation

- Bubble Sort:
    - Outer loop definition/traversal [1]
    - Inner loop definition/traversal [1]
    - Swap condition (must be applied to mean temperature value) [1]
    - Actual element swap (must be applied to entire element) [1]
    - Swap check optimisation [1]

Question 1

Task 1.3 Solution – Evidence 3

```
def getMedian(sortedData):
    days = len(sortedData)
    if days % 2 == 1:
        return float(sortedData[days // 2][6])
    else:
        return ((float(sortedData[days // 2][6]) + \
                float(sortedData[(days // 2) - 1][6])) / 2)

print("Difference in Median Temperatures (June 2019 - June 2018): " + \
      str(getMedian(sortedData19) - getMedian(sortedData18)) + "°C")
```

Task 1.3 – Evidence 3 Mark Allocation

- Calls to Bubble Sort function from Task 1.2 [1]
- Calculating median [1]
- Printing the median based on specification [1]

Question 1

Task 1.4 Solution – Evidence 4

```
def binarySearch(sortedData, temp):
    start = 0
    end = len(sortedData) - 1
    while start != end: # we wish to search till we exhaust all elements
        mid = (start + end) // 2
        if float(sortedData[mid][6]) <= temp:
            start = mid + 1 # sortedData[mid] and below invalid
        else:
            end = mid # sortedData[mid] is possibly the target
    if float(sortedData[start][6]) > temp:
        return start
    else:
        return -1
```

Task 1.4 – Evidence 4 Mark Allocation

- Binary search:
    - Start and end initialisation, and mid calculation in each iteration [1]
    - While loop condition [1]
    - Update of start and end:
        - Mid element versus target check (condition) [1]
        - Update if above condition true [1]
        - Update if above condition false [1]
    - Final condition for no such value found and return statements [1]

Question 1

Task 1.4 Solution – Evidence 5

```
def printDataAboveTemp(data):
    print("{0:<8}{1:<8}{2:<8}{3:<8}{4:<8}".format( \
            "YEAR", "DATE", "RAIN", "TEMP", "WIND"))
    for i in range(len(data) - 1, -1, -1):
        print("{0:<8}{1:<8}{2:<8}{3:<8}{4:<8}".format( \
            data[i][0], data[i][1], data[i][2], data[i][6], data[i][9]))

sortedDataAll = bubbleSort(data18 + data19)

while True:
    temp = float(input("\nPlease input a mean temperature threshold" + \
                       "(°C): "))
    targetIndex = binarySearch(sortedDataAll, temp)
    if targetIndex == -1:
        print("None")
    else:
        printDataAboveTemp(sortedDataAll[targetIndex:])
```

Task 1.4 – Evidence 5 Mark Allocation

- Loop based on specification and the input statement (note: ECF for ° missing/incorrect) [1]
- Calling the Modified Binary Search function to perform the requested searches [1]
- Printing the outcome of the search (output):
    - Loop to print all records satisfying search criteria [1]
    - Format as specified [1]

Question 2

Task 2.1 Solution – Evidence 6

```
def insertionSort(L):
    return insertionSortOuter(L, 0)

def insertionSortOuter(L, i):
    if i >= len(L):
        return L
    else:
        return insertionSortOuter(insertionSortInner(L, i), i + 1)

def insertionSortInner(L, j):
    if j < 1:
        return L
    else:
        if L[j] < L[j - 1]:
            L[j - 1], L[j] = L[j], L[j - 1]
        return insertionSortInner(L, j - 1)
```

Task 2.1 – Evidence 6 Mark Allocation

- Parameters for the call to insertSortOuter function in the return statement for the insertionSort function [1]
- If condition for the insertSortOuter function [1]
- Return statement for the inserSortOuter function [1]
- Outer if condition for the inserSortInner function [1]
- If condition and body for the else part of the outer if statement in the insertSortInner function [1]
- Return statement at the end of the else part of the insertSortInner function [1]

Question 2

Task 2.2 Test Cases – Evidence 7

```
# INPUT     PURPOSE                          EXPECTED OUTPUT
# []        Boundary Case - Zero Elements    []
# [1]       Boundary Case - One Element      [1]
# [1,2,3]   Normal Case - Sorted List        [1,2,3]
# [3,2,1]   Normal Case - Reversed List      [1,2,3]
# [2,1,3]   Normal Case - Randomised List    [1,2,3]
```

Task 2.3 – Evidence 7 Mark Allocation

- Any one valid boundary case (including purpose and expected output) [1]
- Any two valid normal cases (including purpose and expected output) [1]

Question 2

Task 2.3 Solution – Evidence 8

```
def fibSeq(n):
    if n == 0: # n is F0
        return 0
    elif n == 1: # n is F1 or F2
        return 1
    else: # check F3 and above
        return fibSeqRec(1, 1, 3, n)

def fibSeqRec(fa, fb, i, n):
    if fa + fb == n:
        return i
    elif fa + fb > n:
        return -1
    else:
        return fibSeqRec(fb, fa + fb, i + 1, n)
```

Task 2.3 – Evidence 8 Mark Allocation

- Correct main function (wrapper) taking only the parameter n [1]
- Wrapper function:
    - F_0 base case condition and return value [1]
    - F_1 base case condition and return value [1]
    - Recursive call return statement [1]
- Main recursive function:
    - Base case where n = F_k is found, and its appropriate return value [1]
    - Base case where n is not a Fibonacci Number and its appropriate return value [1]
    - Recursive case statement [1]
- Accurate sequence index or else -1 returned [1]

Question 2

Task 2.4 Test Cases – Evidence 9

```
# INPUT     PURPOSE                                   EXPECTED OUTPUT
# 0         Boundary Case - First Base Case           0
# 1         Boundary Case - Second Base Case          1
# 2         Normal Case - Initial Recursive Case      3
# 8         Normal Case - Rec. Case - Is Fib.         6
# 9         Normal Case - Rec. Case - Is not Fib.     -1
```

Task 2.4 – Evidence 9 Mark Allocation

- Any one valid boundary case (including purpose and expected output) [1]
- Any two valid normal cases (including purpose and expected output) [1]

Question 2

Task 2.5 Solution – Evidence 10

```
v1 = 1346270
v2 = 24157817
print(str(v1) + ": " + str(fibSeq(v1)))
print(str(v2) + ": " + str(fibSeq(v2)))
```

Task 2.5 – Evidence 10 Mark Allocation

- Call for 1346270 [1]
- Call for 24157817 [1]

Question 3

Task 3.1 Solution – Evidence 11

```
class PhoneNum(str):

    def __hash__(self):
        hashValue = 0
        for i in range(len(self)):
            hashValue += (i + 1) * ord(self[i])
        return hashValue
```

Task 3.1 – Evidence 11 Mark Allocation

- Class definition that inherits off str [1]
- Loop definition and return value for the Hash method [1]
- Hash value calculation for each character [1]

Question 3

Task 3.2 Solution – Evidence 12

```
class Node():

    def __init__(self):
        self._data = PhoneNum("")
        self._link1 = -1
        self._link2 = -1
        self._link3 = -1

    def getData(self):
        return self._data

    def setData(self, newData):
        self._data = newData

    def getLink1(self):
        return self._link1

    def setLink1(self, newLink1):
        self._link1 = newLink1

    def getLink2(self):
        return self._link2

    def setLink2(self, newLink2):
        self._link2 = newLink2

    def getLink3(self):
        return self._link3

    def setLink3(self, newLink3):
        self._link3 = newLink3

    def print(self):
        print("DATA: " + str(self._data) + "; HASH: " + \
              str(hash(self._data)) + "; LINK1: " + str(self._link1) + \
              "; LINK2: " + str(self._link2) + "; LINK3: " + \
              str(self._link3))
```

Task 3.2 – Evidence 12 Mark Allocation

- Node class init method [1]
- Node get methods (all) [1]
- Node set methods (all) [1]
- Node print method (with specified formatting) [1]

Question 3

Task 3.3 Solution – Evidence 13

```
class HDS():

    def __init__(self, size):
        self._nodes = [Node() for i in range(size)]
        for i in range(len(self._nodes) - 1): # excluding last node
            self._nodes[i].setLink1(i + 1)
        self._free = 0
        self._first = -1

    def print(self):
        for i in range(len(self._nodes)):
            self._nodes[i].print()
```

Task 3.3 – Evidence 13 Mark Allocation

- HDS class init method:
    - Initialisation of the nodes attribute (with correct array size) [1]
    - Initialisation of the link1 values in each Node instance in the nodes attribute [1]
    - Initialisation of the free and first attributes [1]
- HSD class print method:
    - Iteration through each Node instance in the nodes attribute [1]
    - Calling the print method for each Node instance in the nodes attribute [1]

Question 3

Task 3.4 Solution – Evidence 14

```
    def BSTInsert(self, rootNode, rootIndex, newNode, newIndex):
        # 5. BST nodes are sorted in terms of Hash Value
        currentIndex = rootIndex
        currentNode = rootNode
        while True:
            if hash(newNode.getData()) < hash(currentNode.getData()):
                if currentNode.getLink2() == -1:
                    currentNode.setLink2(newIndex)
                    newNode.setLink1(currentIndex)
                    break
                else:
                    currentIndex = currentNode.getLink2()
                    currentNode = self._nodes[currentIndex]
            else:
                if currentNode.getLink3() == -1:
                    currentNode.setLink3(newIndex)
                    newNode.setLink1(currentIndex)
                    break
                else:
                    currentIndex = currentNode.getLink3()
                    currentNode = self._nodes[currentIndex]

    def insert(self, newPhoneNum):
        # 1. remove node at free pointer; update free pointer (free LL)
        if self._free == -1
            return print("Unable to insert. HDS full.")
        newIndex = self._free
        newNode = self._nodes[newIndex]
        self._free = newNode.getLink1()
        # 2. set new node data to newPhoneNum
        newNode.setData(newPhoneNum)
        newNode.setLink1(-1)
        # 3. if HDS is empty
        if self._first == -1:
            # -> insert new node as first LL node in HDS
            self._first = newIndex
        else:
            # 4a. search LL part of HDS for existing country code
            targetCountry = newPhoneNum.split()[0]
            currentIndex = self._first
            currentNode = self._nodes[currentIndex]
            while currentNode.getData().split()[0] != targetCountry and \
                  currentNode.getLink1() != -1:
                currentIndex = currentNode.getLink1()
                currentNode = self._nodes[currentIndex]
            # 4b. if an LL node (t) is found with the same country code
            if currentNode.getData().split()[0] == targetCountry:
                # -> insert new node into the BST with root (t)
                self.BSTInsert(currentNode, currentIndex, \
                               newNode, newIndex)
            # 4c. else if it does not exist in LL part of HDS
            else:
                # -> then insert node at the end of LL part of HDS
                currentNode.setLink1(newIndex)
```

Task 3.4 – Evidence 14 Mark Allocation

- HDS class insert method:
    - Check if HDS is full [1]
    - Get new node to insert from free LL [1]
    - Update free attribute [1]
    - Update attributes for selected node [1]
    - Check if HDS empty; insert as first if necessary [1]
    - Search LL nodes for existing country code:
        - Loop condition [1]
        - Loop iteration (and iterator update) [1]
    - Condition to determine if to be inserted as LL node (else triggers BST insert at LL Node with same country code) [1]
    - Insertion as LL node [1]
    - BST insertion:
        - Iteration through the BST (to search for insertion point) [1]
        - Left child insertion or else traversal [1]
        - Right child insertion or else traversal [1]
        - Traversal of BST based on hash value [1]

Question 3

Task 3.5 Solution – Evidence 15

```
def BSTContains(self, rootNode, targetPhoneNum):
    currentNode = rootNode
    targetHash = hash(PhoneNum(targetPhoneNum))
    while True:
        if currentNode.getData() == targetPhoneNum:
            return True
        if targetHash < hash(currentNode.getData()):
            if currentNode.getLink2() == -1:
                return False
            else:
                currentNode = self._nodes[currentNode.getLink2()]
        else:
            if currentNode.getLink3() == -1:
                return False
            else:
                currentNode = self._nodes[currentNode.getLink3()]

def contains(self, targetPhoneNum):
    if self._first == -1:
        return False
    else:
        # check for existance of country code
        targetCountry = targetPhoneNum.split()[0]
        currentIndex = self._first
        currentNode = self._nodes[currentIndex]
        while currentNode.getData().split()[0] != targetCountry and \
              currentNode.getLink1() != -1:
            currentIndex = currentNode.getLink1()
            currentNode = self._nodes[currentIndex]
        if currentNode.getData().split()[0] == targetCountry:
            return self.BSTContains(currentNode, targetPhoneNum)
        else:
            return False
```

Task 3.5 – Evidence 15 Mark Allocation

- HDS class contains method:
  - Check if empty and return appropriate value if so [1]
  - Iterate through all the LL nodes to check if target (country code) exists [1]
  - Returns False if target country code not found [1]
  - Iteration through BST to check (if LL node with target country code exists):
    - Condition to check if target number matches number in current node (separate from traversal if checks) [1]
    - Left and right child iteration [1]
    - Traversal of BST based on hash value [1]
  - Return True/False appropriately for all cases [1]

Question 3

Task 3.6 Solution – Evidence 16

```
d = HDS(25)
fHandle = open("PHONENUMS.TXT")
for line in fHandle:
    d.insert(PhoneNum(line.strip()))
fHandle.close()
d.print()
```

Task 3.6 – Evidence 16 Mark Allocation

- File I/O and parsing of data (removal of "\n" character) [1]
- Initialisation of HDS as specified, and using the insert method to populate the HDS [1]
- Calling HDS.print() [1]

Question 3

Task 3.6 Screenshot – Evidence 17

```
DATA: 61 94770276; HASH: 3429; LINK1: 2; LINK2: 1; LINK3: 3
DATA: 61 93575117; HASH: 3394; LINK1: 0; LINK2: 9; LINK3: -1
DATA: 65 91388431; HASH: 3392; LINK1: -1; LINK2: 11; LINK3: 5
DATA: 61 94746383; HASH: 3442; LINK1: 0; LINK2: -1; LINK3: 4
DATA: 61 94590685; HASH: 3466; LINK1: 3; LINK2: -1; LINK3: 6
DATA: 65 92646257; HASH: 3439; LINK1: 2; LINK2: 12; LINK3: 8
DATA: 61 95903579; HASH: 3481; LINK1: 4; LINK2: 10; LINK3: 7
DATA: 61 97669278; HASH: 3525; LINK1: 6; LINK2: 14; LINK3: -1
DATA: 65 94449889; HASH: 3567; LINK1: 5; LINK2: 18; LINK3: -1
DATA: 61 95804070; HASH: 3339; LINK1: 1; LINK2: 15; LINK3: -1
DATA: 61 93927809; HASH: 3474; LINK1: 6; LINK2: -1; LINK3: -1
DATA: 65 92107504; HASH: 3333; LINK1: 2; LINK2: 13; LINK3: 17
DATA: 65 95728552; HASH: 3434; LINK1: 5; LINK2: 16; LINK3: -1
DATA: 65 93149012; HASH: 3325; LINK1: 11; LINK2: -1; LINK3: -1
DATA: 61 91959259; HASH: 3497; LINK1: 7; LINK2: -1; LINK3: -1
DATA: 61 97102500; HASH: 3266; LINK1: 9; LINK2: -1; LINK3: -1
DATA: 65 93123385; HASH: 3393; LINK1: 12; LINK2: -1; LINK3: -1
DATA: 65 98110408; HASH: 3349; LINK1: 11; LINK2: 19; LINK3: -1
DATA: 65 91238989; HASH: 3534; LINK1: 8; LINK2: -1; LINK3: -1
DATA: 65 98130461; HASH: 3346; LINK1: 17; LINK2: -1; LINK3: -1
DATA: ; HASH: 0; LINK1: 21; LINK2: -1; LINK3: -1
DATA: ; HASH: 0; LINK1: 22; LINK2: -1; LINK3: -1
DATA: ; HASH: 0; LINK1: 23; LINK2: -1; LINK3: -1
DATA: ; HASH: 0; LINK1: 24; LINK2: -1; LINK3: -1
DATA: ; HASH: 0; LINK1: -1; LINK2: -1; LINK3: -1
>>>
```

Task 3.6 – Evidence 17 Mark Allocation

- Valid screenshot (all link values correct based on insertion order – i.e., as above) [1]

Question 3

Task 3.7 Solution – Evidence 18

```
    def BSTinOrderPrint(self, currentNode):
        if currentNode.getLink2() != -1:
            self.BSTinOrderPrint(self._nodes[currentNode.getLink2()])
        print(currentNode.getData() + " " + \
              str(hash(currentNode.getData())))
        if currentNode.getLink3() != -1:
            self.BSTinOrderPrint(self._nodes[currentNode.getLink3()])

    def orderedPrint(self):

        currentNode = self._nodes[self._first]
        while True:
            self.BSTinOrderPrint(currentNode)
            if currentNode.getLink1() == -1:
                break
            else:
                currentNode = self._nodes[currentNode.getLink1()]
```

Task 3.7 – Evidence 18 Mark Allocation

- Printing an adequate message when the HDS is empty [1]
- Traversal of nodes:
    - In-order sequence [1]
    - Printing phone number and its hash value (else it cannot be properly validated) [1]
- Prints all BST nodes in each BST in order of hash value [1]

Question 4

Task 4.1 Solution – Evidence 19

```
class HexBoard():

    def __init__(self, n):
        self._board = [["-" for i in range(n)] for j in range(n)]
        # with the above, a cell is referenced via self._board[row][col]
        self._turn = 0
        self._n = n

    def playAt(self, row, col, symbol):
        if self._board[row][col] != "-":
            print("Invalid move; row " + str(row) + ", col " + \
                  str(col) + " is not empty.")
        else:
            self._board[row][col] = symbol

    def playX(self, row, col):
        self.playAt(row, col, "X")

    def playO(self, row, col):
        self.playAt(row, col, "O")

    def printBoard(self):
        for row in range(self._n):
            for col in range(self._n):
                print(self._board[row][col], end = " ")
            print()
```

Task 4.1 – Evidence 19 Mark Allocation

- HexBoard class init method (must include both points below) [1]
    - Initialisation of the board (with appropriate number of rows and columns)
    - Initialisation of turn attribute, with appropriate values for all cells in the board attribute, and an appropriate value for the turn attribute
- HexBoard class playX and play methods:
    - Check if the target cell is already empty before allowing to play at that cell (with appropriate warning message) [1]
    - Cell update as specified (with the correct symbol) [1]
- HexBoard class print method [1]

Question 4

Task 4.2 Solution – Evidence 20

```
    def getAdjacentCells(self, row, col, symbol):
        adjacentCells = []
        if col - 1 >= 0 and self._board[row][col - 1] == symbol:
            adjacentCells.append((row, col - 1))
        if row + 1 < self._n and col - 1 >= 0 and \
           self._board[row + 1][col - 1] == symbol:
            adjacentCells.append((row + 1, col - 1))
        if row + 1 < self._n and self._board[row + 1][col] == symbol:
            adjacentCells.append((row + 1, col))
        if row - 1 >= 0 and self._board[row - 1][col] == symbol:
            adjacentCells.append((row - 1, col))
        if row - 1 >= 0 and col + 1 < self._n and \
           self._board[row - 1][col + 1] == symbol:
            adjacentCells.append((row - 1, col + 1))
        if col + 1 < self._n and self._board[row][col + 1] == symbol:
            adjacentCells.append((row, col + 1))
        return adjacentCells

    def traverse(self, traversing, symbol):
        visited = []
        while len(traversing) > 0:
            current = traversing.pop()
            if symbol == "X":
                i = 1 # column index - for "X"
            else:
                i = 0 # row index - for "O"
            if current[i] == self._n - 1: # winning condition
                return True
            if current not in visited:
                visited.append(current)
            adjacentCells = self.getAdjacentCells(current[0], \
                                                  current[1], symbol)
            for i in range(len(adjacentCells)):
                if adjacentCells[i] not in visited and \
                   adjacentCells[i] not in traversing:
                    traversing.append(adjacentCells[i])
        return False

    def checkWinX(self): # left to right
        traversing = []
        for row in range(len(self._board)):
            if self._board[row][0] == "X":
                traversing.append((row, 0))
        return self.traverse(traversing, "X")

    def checkWinO(self): # top to bottom
        traversing = []
        for col in range(len(self._board)):
            if self._board[0][col] == "O":
                traversing.append((0, col))
        return self.traverse(traversing, "O")
```

Task 4.2 – Evidence 20 Mark Allocation

- General strategy to check for win condition for both players established (at least in comments) [1]
- X player win condition [1]
- Y player win condition [1]
- Traversal to check for winning conditions:
    - List of cells on either end (required for a win) [1]
    - Cycle resolution (by checking visited nodes) [1]
    - Condition to check for adjacent nodes (for traversal):
        - Nodes left, right [1]
        - Nodes above, below [1]
        - Node below-and-left [1]
        - Node above-and-right [1]
    - Only adding nodes to the traversal list if they are of the same type [1]
- Check if any traversed nodes are on the other end (from start side) [1]
- Accurate return value [1]

Question 4

Task 4.3 Solution – Evidence 21 (Part 1 of 2)

```
# CASE 1:
b1 = HexBoardv2(3);
b1.playX(0,1);
b1.playX(1,0);
b1.playX(2,0);
b1.playO(0,2);
b1.playO(1,1);
b1.playO(2,1);
b1.printBoard();
print("-> X win: " + str(b1.checkWinX()));
print("-> O win: " + str(b1.checkWinO()));
print("\n")
```

```
- X O
X O -
X O -
-> X win: False
-> O win: True
```

```
# CASE 2:
b2 = HexBoardv2(3)
b2.playX(0,0)
b2.playX(0,1)
b2.playX(0,2)
b2.playO(1,0)
b2.playO(1,1)
b2.playO(1,2)
b2.printBoard()
print("-> X win: " + str(b2.checkWinX()))
print("-> O win: " + str(b2.checkWinO()))
print("\n")
```

```
X X X
O O O
- - -
-> X win: True
-> O win: False
```

```
# CASE 3:
b3 = HexBoardv2(3)
b3.playX(0,0)
b3.playX(1,0)
b3.playX(1,1)
b3.playX(2,1)
b3.playX(2,2)
b3.playO(0,1)
b3.playO(0,2)
b3.playO(1,2)
b3.playO(2,0)
b3.printBoard()
print("-> X win: " + str(b3.checkWinX()))
print("-> O win: " + str(b3.checkWinO()))
print("\n")
```

```
X O O
X X O
O X X
-> X win: True
-> O win: False
```

Question 4

Task 4.3 Solution – Evidence 21 (Part 2 of 2)

```
# CASE 4:
b4 = HexBoardv2(3)
b4.playX(0,0)
b4.playX(0,1)
b4.playX(1,0)
b4.playX(1,2)
b4.playX(2,0)
b4.playO(0,2)
b4.playO(1,1)
b4.playO(2,1)
b4.playO(2,2)
b4.printBoard()
print("-> X win: " + str(b4.checkWinX()))
print("-> O win: " + str(b4.checkWinO()))
print("\n")
```

```
X X O
X O X
X O O
-> X win: False
-> O win: True
```

Task 4.3 – Evidence 21 Mark Allocation

- Case 1 code and output [1]
- Case 2 code and output [1]
- Case 3 code and output [1]
- Case 4 code and output [1]