1. You are provided with the usernames and passwords of some members of a recreation club in the text file `ACCOUNTS.txt`. The format of each record is:

```
<username> <password>
```

Sample records from the file:

```
bzkoh F9obSbf2&
pdnathan %57+g/J[
```

Members use their usernames and passwords to login to the club's portal to book facilities. Arising from a new IT security policy, all passwords must fulfil all the following requirements:

- At least 8 characters in length,
- At least 1 uppercase letter,
- At least 1 lowercase letter,
- At least 1 digit,
- At least 1 punctuation mark / special symbol which is a printable ASCII character (excluding whitespace).

  These characters have ASCII code (in decimal) from

  - 33 to 47
  - 58 to 64
  - 91 to 96
  - 123 to 126

  For examples, ASCII code 33 is '`!`', ASCII code 47 is '`/`'

Examples of passwords that fulfil all requirements:

`AbCd35*n`	`iloveApples2\`	`3#$%^&*(Kc`	`<jumping9Jac>`

Examples of passwords that do not meet all requirements and reasons:

`cocoN8%`	Less than 8 characters.
`coconut8%`	No uppercase letter.
`COCO8888`	No lowercase letter. No symbol
`iLikeDo^`	No digit.
`EatLiv3`	Less than 8 characters. No symbol.
`comeHome`	No digit. no symbol.

Task 1.1

Write program code to find all the passwords from the data file `ACCOUNTS.txt` that do not meet the IT security requirements.

Display this list of passwords together with its usernames and reasons for not meeting requirements.

Display also the number of passwords that fail to meet requirements at the end of the listing.

Sample output:

```
Username        Password      Reasons for not meeting requirements
chiangyy        cocoN8%       Less than 8 characters.
ngpingyi        coconut8%     No uppercase letter.
joegomez        COCO8888      No lowercase letter. No symbol.
michaelbb       iLikeDo^      No digit.
douglaslim      EatLiv3       Less than 8 characters. No symbol.
yusufahmad      comeHome      No digit. No symbol.

6 passwords do not meet requirements
```

Evidence 1: Your program code for **task 1.1**. **[14]**

Evidence 2: Screenshot of running **task 1.1**. **[1]**

[Total: 15 marks]

2. A Fibonacci series is a series of integers in which each number (Fibonacci number) is the sum of the two preceding numbers. This is the start of the Fibonacci series:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, ......
```

The first two terms are both 1, and subsequent term is sum of previous two terms.

Using `Fibonacci(n)` to define the n^{th} term of the Fibonacci series,

```
Fibonacci(1) = Fibonacci(2) = 1                          , n = 1,2
Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2), n = 3,4,5,......
```

Examples:
```
Fibonacci(3) = Fibonacci(2) + Fibonacci(1) = 2
Fibonacci(4) = Fibonacci(3) + Fibonacci(2) = 3
......
Fibonacci(9) = Fibonacci(8) + Fibonacci(7) = 34
```

Task 2.1

Write program code for a **recursive function** `fib` to compute the n^{th} term of the Fibonacci series.

```
FUNCTION fib (n : INTEGER) : INTEGER
```

Write additional code to print the first 15 Fibonacci numbers using `fib`.

Evidence 3: Your program code for **task 2.1** and screenshot of running task. **[5]**

Task 2.2

Write program code for a **non-recursive function** `fib_nr` to compute the n^{th} term of the Fibonacci series.

```
FUNCTION fib_nr (n : INTEGER) : INTEGER
```

Write additional code to print the first 15 Fibonacci numbers using `fib_nr`.

Evidence 4: Your program code for **task 2.2** and screenshot of running task. **[5]**

Task 2.3

Recursive function `fib` will call itself when it is run.

Amend your recursive function `fib` to ***show the number of times*** that the function `fib` is called *from the main program* for a given value of `n`.

Test your amended program code by calling `fib(10)` and `fib(30)` to show the number of times that `fib` is called when `n=10` and `n=30`.

Evidence 5: Your amended program code for `fib`. Screenshots of running `fib` for `n=10` and `n=30`. **[3]**

Evidence 6: For any given value of `n`, function `fib` takes a longer time to run than `fib_nr`. Explain why. **[2]**

[Total: 15 marks]

3. Write a program to find all the words in a text and print them in alphabetical order, along with its number of occurrences. Use a linked list to store this information. Each node of the linked list will hold a word, the number of occurrence of that word, and a pointer to the next node in alphabetical order.

Implement each node as an instance of the class `Node`, which has the following properties:

Class: `Node`		
Properties		
Identifier	**Data Type**	**Description**
`Word`	`STRING`	The node's value for the word
`Count`	`INTEGER`	The node's value for the number of occurrences of that word
`Pointer`	`INTEGER`	The pointer of the node

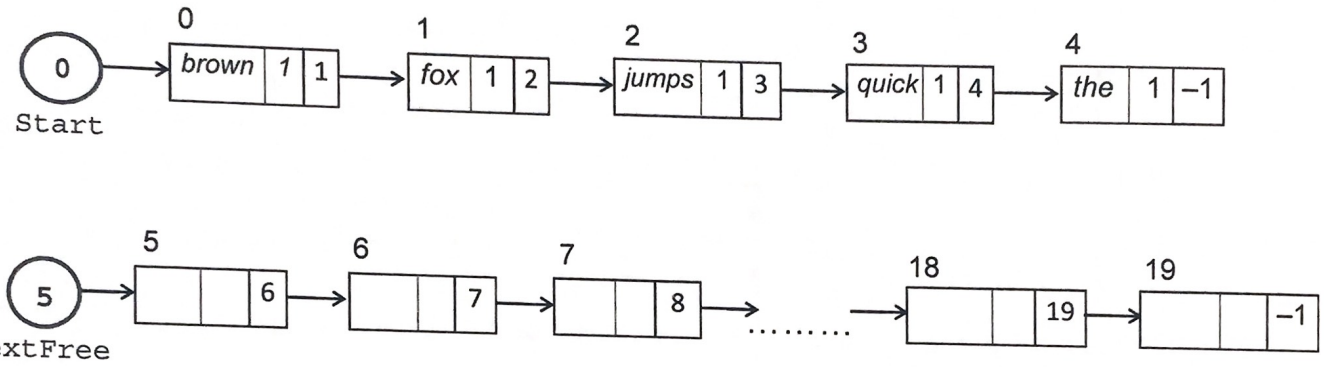
Implement the linked list using an instance of the class `LinkedList`, which has the following properties and methods:

Class: `LinkedList`		
Properties		
Identifier	**Data Type**	**Description**
`WordList`	`ARRAY[20] of Node`	The words and number of occurrences from the text stored in an array of 20 nodes
`Start`	`INTEGER`	Index for the root position of the `WordList` array
`NextFree`	`INTEGER`	Index for the next unused node
Methods		
`Initialise`	`PROCEDURE`	Sets all node data values to empty string (for `Word`) and 0 (for `Count`). Set pointers to indicate all nodes are unused and linked. Initialise values for `Start` and `NextFree`.
`Display`	`PROCEDURE`	Display the current state of array content and pointers in table index form.
`Insert`	`PROCEDURE`	Insert word from the text into the linked list. If word exists in linked list, increment `Count` by 1. If word does not exist in linked list, add new node with new word and set `Count` to 1.

Maintain a linked list of all the unused nodes. Indicate `NULL` with integer -1.

The diagram shows the linked list with:

- the words "`the quick brown fox jumps`" added
- the unused nodes linked together.



Task 3.1

Write the program code for the classes `Node` and `LinkedList`, including the `Initialise` and `Display` method. The code should follow the specifications given. Do not write the `Insert` procedure yet.

Evidence 7: Your program code for the `Node` and `LinkedList` classes. **[10]**

Task 3.2

Write program code in the **main program** to create a linked list object and display the linked list in index order.

Evidence 8: **Program code** and **screenshot** confirming all values after initialisation of the `LinkedList` object. **[3]**

Task 3.3

Write the `Insert` procedure that will insert a word from the text into the linked list.

If word already exists in linked list, increment `Count` by 1.

If word does not exist in linked list, add a new node with the new word and set `Count` to 1. You should check for availability of free node before adding a new node.

If you use additional method(s) and/or variable(s), present them in a table like this:

Additional method / variable	Description
... ...	... ...

Add appropriate comments to explain your program code.

Evidence 9: Your program code for `Insert` procedure and other methods (if any), **with suitable comments**. Table of additional method(s) and/or variable(s), if any. **[17]**

Task 3.4

Write additional code in the **main program** to read all the words from the text file `WORDTEXT.txt` and display the linked list in index order.

Evidence 10: Your **additional program code** and **screenshot** showing the contents of the linked list after reading in all the words from the text file `WORDTEXT.txt`. **[4]**

Task 3.5

Write a **recursive** `ReverseTraversal` procedure that will traverse the linked list in reverse order and display the words and its count.

Evidence 11: Your recursive `ReverseTraversal` **program code** and **screenshot** of running the procedure. **[6]**

[Total: 40 marks]

4. Records of songs are stored in a hash table. Each song record is an instance of the class `SongRecord`, which has `SongID` and `SongTitle` as its data members.

Implement the hash table as an instance of the class `HashTable` using the following properties and methods:

<table>
<tr><th colspan="2">Class: <code>HashTable</code></th></tr>
<tr><td colspan="2">Properties</td></tr>
<tr><td>Identifier</td><td>Description</td></tr>
<tr><td><code>Size</code></td><td>Maximum number of addresses in hash table</td></tr>
<tr><td><code>Array</code></td><td>One-dimensional array of hash table to store records indexed <code>0</code> to <code>(Size - 1)</code></td></tr>
<tr><td colspan="2">Methods</td></tr>
<tr><td><code>Initialise</code></td><td>Initialises <code>Size</code> and <code>Array</code>.</td></tr>
<tr><td><code>Hash</code></td><td>• Function to calculate the address of the hash table
• Takes <code>SongID</code> as argument
• ASCII code is calculated for each character from <code>SongID</code>
• Total of all ASCII values is calculated
• Total is divided by <code>Size</code> and the remainder calculated with modulo arithmetic, where <code>Size</code> is the maximum number of locations
• Value returned by the function is <code>remainder</code>. This value is the address for the record in the hash table</td></tr>
<tr><td><code>Display</code></td><td>• Displays the contents of the hash table under this heading:
<table><tr><th>Index</th><th>Song ID</th><th>Song Title</th></tr><tr><td>...</td><td>...</td><td>...</td></tr></table></td></tr>
<tr><td><code>Add</code></td><td>• Adds a record into the hash table
• Takes <code>SongID</code> and <code>SongTitle</code> as arguments</td></tr>
<tr><td><code>Remove</code></td><td>• Removes a record from the hash table
• Takes <code>SongID</code> as argument</td></tr>
</table>

For example, if `Size` has value of `13`, a song record with `SongID` "`T3311`" will be hashed to address `11`.

Total ASCII ("T" + "3" + "3" + "1" + "1") = 284

284 mod 13 = 11

Therefore, add record with `SongID` "`T3311`" to the hash table array with address `11`.

Task 4.1

Write program code for the class `SongRecord`, with data type specified clearly for its data members.

Evidence 12: Program code for class `SongRecord`. **[4]**

Task 4.2

Write program code for the class `HashTable`, using the specifications above. Include all the methods stated: `Initialise`, `Hash`, `Display`, `Add`, and `Remove`.

Assume no collision of records for `Add` and `Remove` here.

Evidence 13: Program code for class `HashTable`. **[14]**

Task 4.3

Test your program code for `Size` of value 13, by

(a) adding the following records into the hash table and **display** the hash table. You may copy and paste these song records from the data file `SONG1.txt`.

Song ID	Song Title
T3311	Titanium
V2233	Victory
H7444	Happy
G5955	Golden

(b) removing song record `H7444` and **display** the hash table.

Evidence 14: Screenshots of running **task 4.3 (a) and (b)**. **[2]**

Task 4.4

Amend your program code to your `HashTable` class to handle collisions when more than one record is hashed to the same location.

Explain how the collision handling works.

When adding a new record, ensure that no record with the same `SongID` is stored in more than one location. Output an appropriate message if an existing song record is added.

When removing a record, output an appropriate message if record does not exist in hash table.

Evidence 15: Amended program code for `HashTable` class.

Highlight your amended program code in **bold**.

Explanation of how collision handling works. **[8]**

Task 4.5

Test your amended program code by

(a) adding the following song records into the hash table. You may copy and paste these song records from the data file `SONG2.txt`.

Song ID	Song Title
G5955	Golden
C9999	Champion
D7474	Delta
J8868	Jump
R1711	Radar

(b) removing song record `S1234` and `G5955`.

Evidence 16: Screenshots of running **task 4.5 (a) and (b)**. **[2]**

[Total: 30 marks]

~~~ END OF PAPER ~~~
~~~