1(a)(i)

x	y	result (before)	rem	result(after)
5	12	0	1	12
2	24	12	0	12
1	48	12	1	60
0				

1(a)(ii)

y	X	result (before)	rem	result(after)
12	5	0	0	0
6	10	0	0	0
3	20	0	1	20
1	40	20	1	60
0				

1(b)(i) Use the last binary digit (bit) of x. E.g. 5_2 = 101 so 5 % 2 = 1. 12_2 = 1100 so 12 % 2 = 0.

1(b)(ii) Truncate x before the last digit. E.g. 5_2 = 101 so 5 // 2 = 10. 12_2 = 1100 so 12 // 2 = 110.

1(c) Boundary: x = 0 or y = 0 (or both)
Erroneous: x is negative

2(a)(i)

1. Considering the array to start from index 1, and there are N elements in the array.
2. Consider array[1] to be sorted.
3. Iterate from array[2] to array[N].
4. Compare the current element (array[i]) to the element just before it (array[i-1]).
5. If the current element is smaller, swap the two elements (swap array[i] with array[i-1]).
6. Repeat step 4 and 5 until the current element is larger than the element just before it.
7. Continue with the iteration.

2(a)(ii) Best case: when the list is already sorted,

Time complexity: O(n)

2(b)(i) line 7: `IF MyList[j] < MyList[j+1]`

2(b)(ii) Add the line after line 13, before line 14:

```
n ← n - 1
```

This reduces the number of iterations for the inner loop. Since the elements after index `n` are already sorted, there is no need for the inner loop to go there.

2(c)(i) Quick sort is done in-place, while merge sort is not in-place, hence quick sort requires less memory

2(c)(ii)

Merge sort has a worst case time complexity of O(NlogN), while quick sort has $O(N^2)$

2(c)(iii)

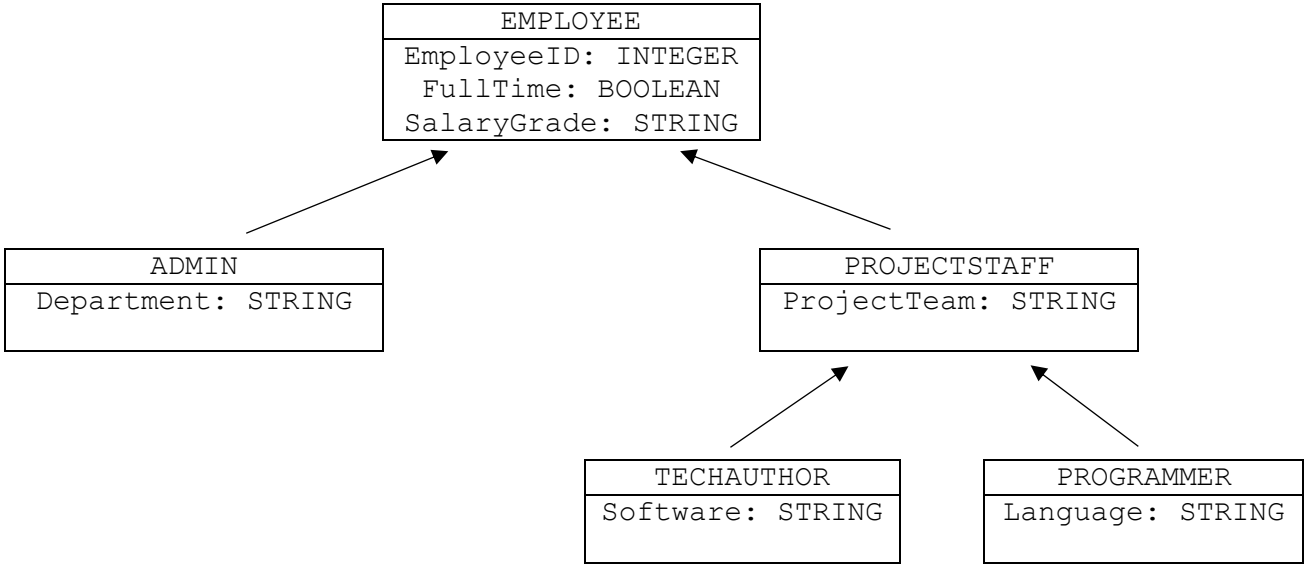
The function calls itself

The function resolves to a base case

3(a)

A class is a blueprint listing the properties and methods common to objects within that class.
An object is an instance of a class.
Many different objects which belong to the same class can be created.

3(b)



3(c)

A subclass inherits properties and methods from a superclass, meaning that when they are defined in the superclass, they do not need to be redefined when the subclass is created. This promotes code reusability and reduces the chances of making a mistake when changes are made.

One example from the class diagram above.

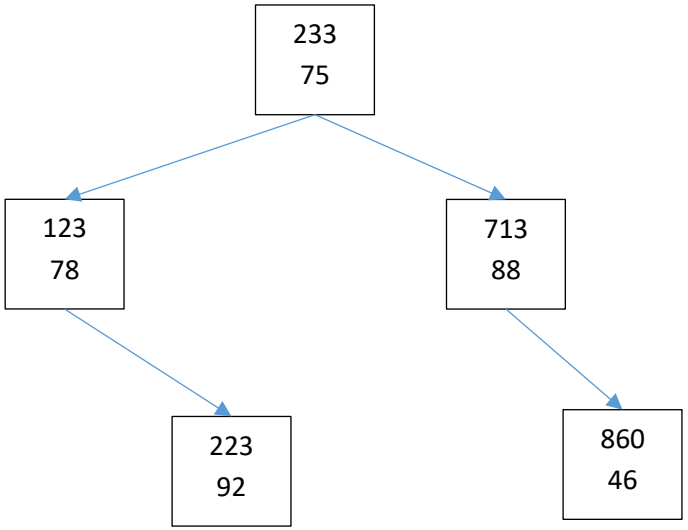
4(a)(i)

Index	Total
1	3
2	11
3	26

4(a)(ii)	26	

4(a)(iii)	`FUNCTION get_value(data_serialno: STRING): INTEGER`
	`  key <- GenerateHash(data_serialno) //1 mark`
	`  RETURN h_table[key][2] // 1 mark`	

4(a)(iv)	O(1) GenerateHash can be considered to be O(1) as the length of the input string is of fixed length. Since accessing an array index is also O(1) (h_table[key][2], the function is O(1)	
4(b)(i)	Any 3 of the below points and a conclusion Hash table has constant search/insert/delete time O(1), while BST has O(lgn), so hash table search is faster Collisions might occur in hash tables that increases the search time, and it could be O(n) in the worst case. BST could have O(n) search time if the tree is unbalanced, while collisions are rare when the hash function is good. BST can get list of sorted items by doing in-order traversal, but not for hash table BST is more memory efficient as it does not require more memory than necessary but hash tables require a lot more memory than required to prevent collisions.	
4(b)(ii)	One 2-D array with 4 columns storing data_serialno, data_score, left_pointer and right_pointer Left pointer stores index value of node that has smaller serialno and right pointer for nodes with larger serialno Variable (named free) to store the index number of the first free node. Variable (named root) to store the index number of the root node. The value -1 (or anything sensible) is used for the null pointers on the terminal nodes (Alternative solution: four 1D arrays or 2 2D arrays with suitable explanation)	

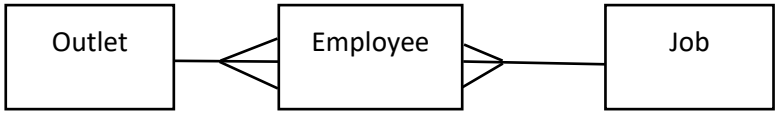
<table>
<tr><td></td><td> 
</td><td></td></tr>
<tr><td>6(c)(i)</td><td>A stack is a linear data structure that follows last in first out and first in last out order.
New data is added to the top, and removed from the top</td><td></td></tr>
<tr><td>6(c)(ii)</td><td>At each recursive call the return address/machine state and return value
Is pushed onto the stack
At the completion of each recursive call
The return address and return value are popped from the stack</td><td></td></tr>
<tr><td>5(a)</td><td>Non-primary key dependency where Salary is not dependent on employeeID but dependent on Job Grade</td><td></td></tr>
<tr><td>5(b)</td><td>Outlet (Primary Key: OutletID)
<table>
<tr><th>OutletID</th><th>Outlet Name</th></tr>
<tr><td>O1</td><td>Gem</td></tr>
<tr><td>O2</td><td>Moon Vista</td></tr>
<tr><td>O3</td><td>East Gate</td></tr>
<tr><td>O4</td><td>LCube</td></tr>
</table>

Employee (Primary Key: EmployeeID)</td><td></td></tr>
</table>

Employee ID	Employee Name	Job Grade	OutletID
E1	James	J1	O1
E2	Sally	J2	O2
E3	Bala	J1	O2
E4	Molly	J1	O1
E5	Ahmad	J2	O4

Job (Primary Key: Job Grade)

Job Grade	Salary
J1	5000
J2	6000
J1	5000
J1	5000
J2	6000



5(c)

```
Outlet(OutletID, OutletName)
Employee(EmployeeID, EmployeeName, OutletID, JobGrade)
Job(JobGrade, Salary)
```

5(d)

```
SELECT EmployeeName [1]
FROM  Employee, Outlet [1]
where (Employee.OutletID = Outlet.OutletID [1]
 and OutletName = "LCube") [1]
```

or

```
SELECT EmployeeName [1]
FROM  Employee [1]
Inner Join Outlet ON Employee.OutletID = Outlet.OutletID [1]
WHERE OutletName = "LCube" [1]
```

6(a)(i) Check that length of password string is at least 8

6(a)(ii) Check that password contains at least one lowercase letter

6(b)

Conditions	At least one capital letter	Y	Y	Y	Y	N	N	N	N
	At least one number	Y	Y	N	N	Y	Y	N	N
	At least one symbol	Y	N	Y	N	Y	N	Y	N
Actions	Vert strong password	X							
	Strong password		X	X		X			
	Weak password				X		X	X	X

6(c) User inputs password twice, server checks if both strings entered are identical before saving it as the user's password

6(d) Since a good hash function is irreversible, even if the database is accessed, the original passwords cannot be recreated from the hashed passwords. This keeps the original passwords secure.

When a user logs in, the entered password is hashed and the result is compared to the hashed password associated with that username in the database. If the two hashed results match, then the user's identity is verified. If they do not match, the user is told to try to login again.

7(a) The URL is sent to the local Domain Name Server (DNS) which looks up the URL both within the server's own database and cache. If the IP address is found, it is sent to the browser. Otherwise, the DNS makes a request to higher level DNS servers until the IP address is found. The local DNS adds it to its cache and the IP address is sent to the browser.

7(b) An IPv4 address consists of 4 bytes, which translates into 4 denary numbers from 0 to 255. The IP address given has only 3 denary numbers and the second one is 299 which is larger than 255.

7(c)(i) The characters in the webpage were encoded using the foreign country's local system of extending ASCII to encode their language's characters. As a result, when the user's browser loads the webpage, it was

	decoded using a different character decoding system, which resulted in meaningless characters. It is also possible that the user's computer did not contain the fonts necessary to load the foreign language's characters.	
7(c)(ii)	Unicode intends to (ultimately) encode all the world's written characters into one universal system so that browsers everywhere can use only one encoding system. This also allows multiple languages on the same page to be displayed simultaneously.	