QUESTION 8.

6 **(a)** A procedure that calls itself // is defined in terms of itself **[1]**

(b) Before procedure call is executed current state of the registers/local variables is saved onto the stack
When returning from a procedure call the registers/local variables are re-instated **[2]**

(c)

Call number	n	(n=0) OR (n=1)	n DIV 2	n MOD 2
1	40	FALSE	20	0
2	20	FALSE	10	0
3	10	FALSE	5	0
4	5	FALSE	2	1
5	2	FALSE	1	0
6	1	TRUE		
		1 mark	*1 mark*	*1 mark*

OUTPUT 101000 – 1 mark for each pair of bits. **[6]**

(d) Conversion of denary number into binary **[1]**

(e) (i) Example Pascal

```
Procedure X(n: INTEGER)
BEGIN
  IF (n = 0) OR (n = 1)
    THEN
        Write(n)
    ELSE
      BEGIN
        X(n DIV 2);
        Write(n MOD 2);
      END;
END;
```

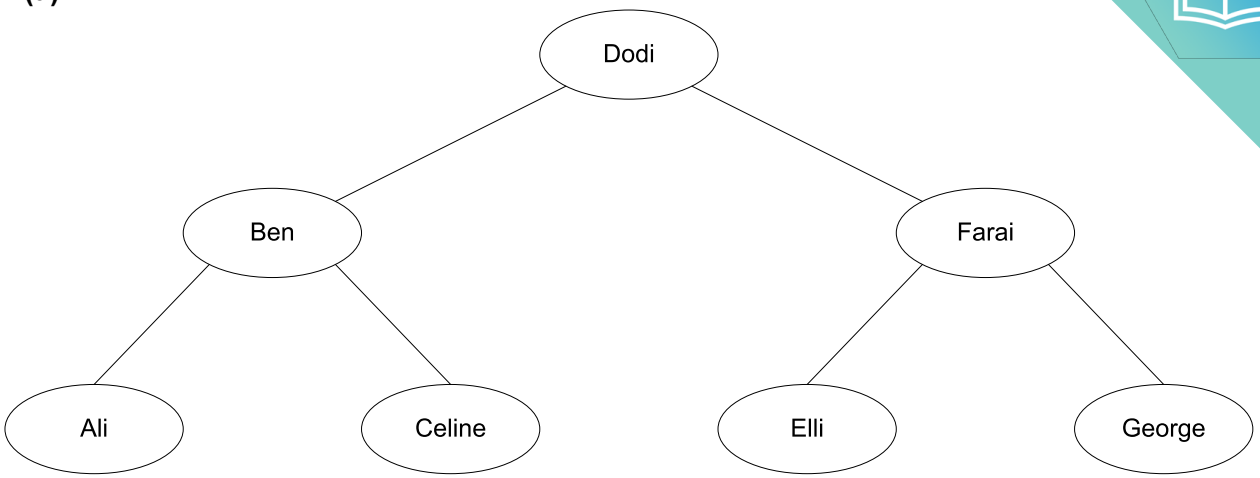
Example Python

```
def X(n):
    if (n == 0) or (n == 1):
       print(n, end="")
    else:
       X(n // 2)
       print(n % 2, end="")
```

Mark as follows:
Procedure heading & ending
Boolean expression
correctly grouped statements within ELSE
recursive call
Using DIV and MOD correctly **[5]**

QUESTION 9.

4 (a)



[4]

(b)

RootPointer: 1

FreePointer: 8

Tree

	Name	LeftPointer	RightPointer
[1]	Dodi	5	2
[2]	Farai	3	4
[3]	Elli	0	0
[4]	George	0	0
[5]	Ben	7	6
[6]	Celine	0	0
[7]	Ali	0	0
[8]		9	0
[9]		10	0
[10]		0	0

[7]

(c) (i)

```
01 PROCEDURE TraverseTree(BYVALUE Root : INTEGER)
02     IF Tree[Root].LeftPointer < > 0
03         THEN
04             TraverseTree(Tree[Root].LeftPointer)
05     ENDIF
06     OUTPUT Tree[Root].Name
07     IF Tree[Root].RightPointer < > 0
08         THEN
09             TraverseTree(Tree[Root].RightPointer)
10     ENDIF
11 ENDPROCEDURE
```

[5]

(ii) A procedure that calls itself // is defined in terms of itself
Line number: 04/09

[2]

QUESTION 10.

2 (a) (i)	A procedure which is defined in terms of itself // A procedure which makes a call to itself // A procedure that calls itself	**1**
(ii)	08 // 8	**1**

Question	Answer	
(b) (i)		

Index	Item
1	**9**
2	
3	
4	
5	
6	
7	
8	

MyList									
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
3	5	8	9	13	16	27	0	0	0
			13						
				16					
					27				
						0			

Note: Final mark only if no additional entries in table
Accept last row to show all final values

(ii)	Any one from: Deletes/removes parameter value/ Item (from the array `MyList`) // Deletes the first entry (in `MyList`) that equals or is bigger than `Item` Overwrites `Item` by moving subsequent items up/down/across/left R right	**1**

QUESTION 11.

Question	Answer	

3	FUNCTION Find(BYVAL Name : STRING,
	BYVAL Start : INTEGER,
	BYVAL Finish : INTEGER) RETURNS INTEGER
	// base case
	IF **Finish < Start** 1
	THEN
	RETURN -1
	ELSE
	Middle ← **(Start + Finish) DIV 2** 1
	IF **NameList[Middle] = Name** 1
	THEN
	RETURN **Middle** 1
	ELSE // general case
	IF SearchItem > **NameList[Middle]** 1
	THEN
	Find(Name, Middle + 1, Finish) 1
	ELSE
	Find(Name, Start, Middle - 1) 1
	ENDIF
	ENDIF
	ENDIF
	ENDFUNCTION	

Question	Answer	Marks

QUESTION 12.

Question	Answer	Marks
2(a)	• A pointer that doesn't point to another node/other data/address // indicates the end of the branch	1
2(b)	one mark per bullet • node with 'Athens' linked to left pointer of Berlin (ignore null pointer) • null pointers in left and right pointers of Athens	2
2(c)(i)	RootPointer: 0 FreePointer: 7 **1 mark**	5
2(c)(ii)	• –1 • It is not the number for any node.	2

	LeftPointer	Tree Data	RightPointer
[0]	**2**	Dublin	**1**
[1]	**-1/∅**	London	**3**
[2]	**6**	Berlin	**5**
[3]	**4**	Paris	**-1/∅**
[4]	**-1/∅**	Madrid	**-1/∅**
[5]	**-1/∅**	Copenhagen	**-1/∅**
[6]	**-1/∅**	Athens	**-1/∅**
[7]	**8**		**-1/∅**
[8]	**9**		**-1/∅**
[9]	**-1/∅**		**-1/∅**

Question	Answer	Marks
2(d)(i)	TYPE Node **LeftPointer : INTEGER** **RightPointer : INTEGER** 1 **Data : STRING** ENDTYPE DECLARE Tree : ARRAY[0 : 9] **OF Node** 1 DECLARE FreePointer : INTEGER DECLARE RootPointer : INTEGER PROCEDURE CreateTree() DECLARE Index : INTEGER **RootPointer ← -1** 1 **FreePointer ← 0** 1 FOR Index ← 0 TO 9 // link nodes **Tree[Index].LeftPointer ← Index + 1** 1 **Tree[Index].RightPointer ← -1** 1 ENDFOR **Tree[9].LeftPointer ← -1** 1 ENDPROCEDURE	**7**

Question	Answer	Marks

2(d)(ii)	PROCEDURE AddToTree(ByVal NewDataItem : STRING)
	// if no free node report an error
	IF FreePointer **= -1** 1
	THEN
	ERROR("No free space left")
	ELSE // add new data item to first node in the free list
	NewNodePointer ← FreePointer
	Tree[NewNodePointer].Data ← NewDataItem 1
	// adjust free pointer
	FreePointer ← **Tree[FreePointer].LeftPointer** 1
	// clear left pointer
	Tree[NewNodePointer].LeftPointer ← **-1** 1
	// is tree currently empty ?
	IF **RootPointer = -1** 1
	THEN // make new node the root node
	RootPointer ← NewNodePointer 1
	ELSE // find position where new node is to be added
	Index ← RootPointer
	CALL FindInsertionPoint(NewDataItem, Index, Direction)
		8

Question	Answer	Marks
	IF Direction = "Left" THEN // add new node on left **Tree[Index].LeftPointer ← NewNodePointer** 1 ELSE // add new node on right **Tree[Index].RightPointer ← NewNodePointer** **1** ENDIF ENDIF ENDIF ENDPROCEDURE	
2(e)	**1 mark per bullet** • test for base case (null/-1) • recursive call for left pointer • output data • recursive call for right pointer • order, visit left, output, visit right IF Pointer <> NULL 1 THEN TraverseTree(Tree[Pointer].LeftPointer) 1 OUTPUT Tree[Pointer].Data 1 + 1 TraverseTree(Tree[Pointer].RightPointer) 1 ENDIF ENDPROCEDURE	**5**

QUESTION 13.

<table>
<tr><th>Question</th><th>Answer</th><th>Marks</th></tr>
<tr><td>4(a)(i)</td><td>A function/subroutine defined in terms of itself // a function/subroutine that calls itself</td><td>1</td></tr>
<tr><td>4(a)(ii)</td><td>06</td><td>1</td></tr>
<tr><td>4(b)</td><td>1 mark for each bullet point:

• –60 as final return value
• 3*2*1*–10

1 mark for each row in table

<table>
<tr><th>Call Number</th><th>Function call</th><th>Number = 0 ?</th><th>Return value</th></tr>
<tr><td>1</td><td>Calculate(3)</td><td>False</td><td>3*Calculate(2)</td></tr>
<tr><td>2</td><td>Calculate(2)</td><td>False</td><td>2*Calculate(1)</td></tr>
<tr><td>3</td><td>Calculate(1)</td><td>False</td><td>1*Calculate(0)</td></tr>
<tr><td>4</td><td>Calculate(0)</td><td>TRUE</td><td>–10</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
</table>
</td><td>6</td></tr>
<tr><td>4(c)(i)</td><td>1 mark per bullet point:

• Each time it calls itself the variables are put onto the stack // The function call itself too many times
• … it runs out of stack space // stack overflow</td><td>2</td></tr>
</table>

Question	Answer	Marks

4(c)(ii)	1 mark per bullet point to max 5:
	
	• Function header with parameter **and** Returning calculated value
	• Loop **parameter** times (up to number, or down from number)…
	• …Multiplying by loop counter
	• Multiplying by –10
	• Dealing with starting value correctly
	
	For example:
	
	`FUNCTION Calculate(Number : INTEGER) RETURNS INTEGER`
	`  DECLARE Count : INTEGER`
	`  DECLARE Value : INTEGER`
	`  Value ← -10`
	`  FOR Count ← 1 to Number`
	`  Value ← Value * Count`
	`  ENDFOR`
	
	`  RETURN Value`
	`ENDFUNCTION`
		5

Question	Answer	Marks
5(a)	1 mark per bullet point to max 2: • To restrict **direct** access to the property to the class // keep the properties secure // So the data can only be accessed by its methods // makes the program more robust • To make the program easier to debug • To ensure data going in is valid // to stop invalid changes // stop accidental changes	**2**

QUESTION 14.

Question	Answer	Marks
5(c)(ii)	1 mark per bullet point to max 2: Example: • If there is any delay to a task which is part of the critical path • … then the overall project will be delayed • Gives the earliest possible completion time • … this allows you to organise/allocate resources (efficiently) • Frequently recalculate the critical path … • … to see if there are any delays/new critical path has arisen • Can identify where there is slack in activities • … so they can start later without affecting the critical path	**2**

Question	Answer	Marks

6(a)(i)	1 mark per bullet point:
	
	• Declaring a record type // class etc.
	• Declaring Country as a string
	• Declaring Pointer as an integer
	
	Example:
	
	`TYPE ListElement`
	`  DECLARE Country : STRING`
	`  DECLARE Pointer : INTEGER`
	`ENDTYPE`
		3

Question	Answer	Marks
6(a)(ii)	1 mark per bullet point: • Declaring `CountryList` as an array with 15 elements • Of type `ListElement` Example: `DECLARE CountryList : ARRAY[1 : 15] OF ListElement`	**2**

6(b)	1 mark for each completed statement
	
	`PROCEDURE DeleteNode(NodeValue: STRING, ThisPointer: INTEGER, PreviousPointer: INTEGER)`
	` IF CountryList[ThisPointer].Value = NodeValue`
	`  THEN`
	`  CountryList[ThisPointer].Value ← ""`
	`  IF ListHead = ThisPointer`
	`  THEN`
	`  ListHead ← CountryList[ThisPointer].Pointer`
	`  ELSE`
	`  CountryList[PreviousPointer].Pointer ← CountryList[ThisPointer].Pointer`
	`  ENDIF`
	`  CountryList[LastNode].Pointer ← ThisPointer`
	`  LastNode ← ThisPointer`
	`  CountryList[ThisPointer].Pointer ← -1`
	`  ELSE`
	`  IF CountryList[ThisPointer].Pointer <> -1`
	`  THEN`
	`  CALL DeleteNode(NodeValue,CountryList[ThisPointer].Pointer, ThisPointer)`
	`  ELSE`
	`  OUTPUT "DOES NOT EXIST"`
	`  ENDIF`
	` ENDIF`
	`ENDPROCEDURE`
		5

QUESTION 15.

Question	Answer	Marks
3(a)	1 mark per bullet point to max 2 • It is defined in terms of itself // it calls itself • It has a stopping condition // base case • It is a self-contained subroutine • It can return data to its previous call	**2**
3(b)	1 mark per bullet point to max 3 • (When the recursive call is made) all values/data are put on … • … the stack • When the stopping condition/base case is met • …the algorithm unwinds • …the last set of values are taken off the stack (in reverse order)	**3**