Question No. 1

e.g. in Python:

```
def BinarySearch(Low, High):                                    [1]
    global Found
    if Low>High:                                        )
        return                                          )       [1]
    Middle=int((High+Low)/2)
    if SearchData[Middle] == SearchItem:                )
        Found = Middle                                  )       [1]
    elif SearchData[Middle] < SearchItem:               )
        BinarySearch(Middle + 1, High)                  )       [1]
    elif SearchData[Middle] > SearchItem:               )
        BinarySearch(Low, Middle - 1)                   )       [1]
    return


BinarySearch(1,63)                                              [1]
```

Question No. 2

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
		7

Question No. 3

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**

Question No. 4

10(a)	**One** mark for each correct marking point **(Max 3)** • Must have a base case/stopping condition • Must have a general case • … which calls itself (recursively) // Defined in terms of itself • … which changes its state and moves towards the base case Unwinding can occur once the base case is reached.	**3**
10(b)	**One** mark for each correct marking point **(Max 3)** • A stack is a LIFO data structure • Each recursive call is pushed onto the stack • …. and is then popped as the function ends • Enables backtracking/unwinding … to maintain the required order.	**3**

Question No. 5

(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]**

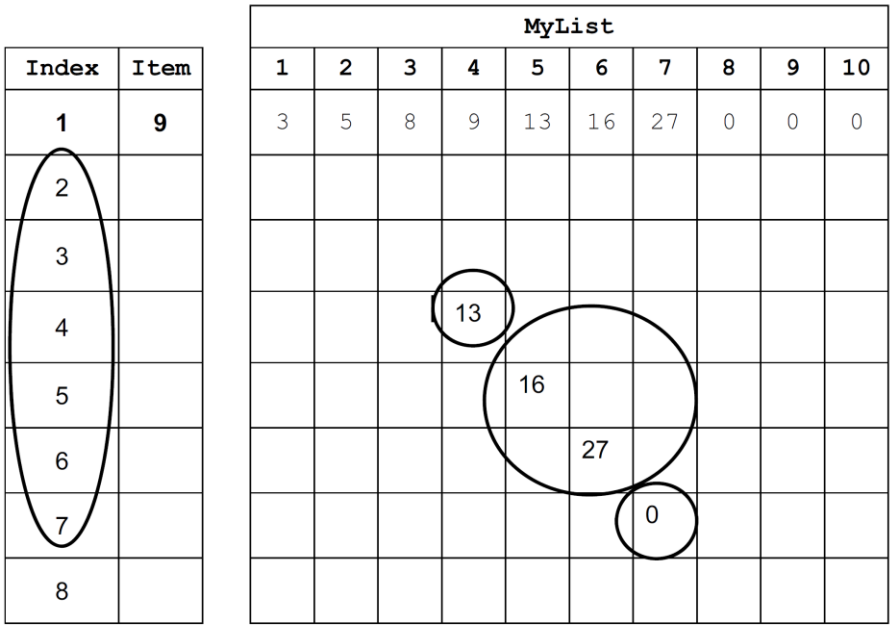
(e)

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 No. 6

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
(b) (i)	 Note: Final mark only if no additional entries in table Accept last row to show all final values	4
(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 No. 7

<table>
<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>
<tr><td>4(c)(ii)</td><td>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:

<pre>
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
</pre></td><td>5</td></tr>
</table>

Question No. 8

<table>
<tr><td>8(a)</td><td>• 8 …
• …it calls itself</td><td>2</td></tr>
<tr><td>8(b)</td><td>1 mark each:

• Final return value = 5
• Output column
• Return value column
• Value 1 and Value 2 columns
• Temp column

<table>
<tr><th>Value1</th><th>Value2</th><th>Temp</th><th>EndValue</th><th>OUTPUT</th><th>Return Value</th></tr>
<tr><td>1</td><td>1</td><td>1</td><td>12</td><td>1</td><td>5</td></tr>
<tr><td>2</td><td></td><td></td><td></td><td>2</td><td>4</td></tr>
<tr><td>3</td><td>2</td><td>2</td><td></td><td>3</td><td>3</td></tr>
<tr><td>5</td><td>3</td><td>3</td><td></td><td>5</td><td>2</td></tr>
<tr><td>8</td><td>5</td><td>5</td><td></td><td>8</td><td>1</td></tr>
<tr><td>13</td><td>8</td><td></td><td></td><td>13</td><td>0</td></tr>
</table></td><td>5</td></tr>
<tr><td>8(c)</td><td>To output/find a value that is the addition of the two previous values // (output) Fibonacci sequence</td><td>1</td></tr>
</table>