OXFORD AQA

INTERNATIONAL QUALIFICATIONS

INTERNATIONAL A-LEVEL

COMPUTER SCIENCE

CS03

Unit 3 Advanced Programming

Mark scheme

Specimen

Version: 1.0 Final

Mark schemes are prepared by the Lead Assessment Writer and considered, together with the relevant questions, by a panel of subject teachers. This mark scheme includes any amendments made at the standardisation events which all associates participate in and is the scheme which was used by them in this examination. The standardisation process ensures that the mark scheme covers the students' responses to questions and that every associate understands and applies it in the same correct way. As preparation for standardisation each associate analyses a number of students' scripts. Alternative answers not already covered by the mark scheme are discussed and legislated for. If, after the standardisation process, associates encounter unusual answers which have not been raised they are required to refer these to the Lead Examiner.

It must be stressed that a mark scheme is a working document, in many cases further developed and expanded on the basis of students' reactions to a particular paper. Assumptions about future mark schemes on the basis of one year's document should be avoided; whilst the guiding principles of assessment remain constant, details will change, depending on the content of a particular examination paper.

Further copies of this mark scheme are available from oxfordaqaexams.org.uk

How to mark

Aims

When you are marking your allocation of scripts your main aims should be to:

- recognise and identify the achievements of students
- where relevant, place students in the appropriate mark band and in the appropriate part of that mark band (high, low, middle) for **each** assessment objective
- record your judgements with brief notes, annotations and comments that are relevant to the mark scheme and make it clear to other associates how you have arrived at the numerical mark awarded for each assessment objective
- ensure comparability of assessment for all students, regardless of question or examiner.

Approach

It is important to be **open minded** and **positive** when marking scripts.

The specification recognises the variety of experiences and knowledge that students will have. It encourages them to study computer science in a way that is relevant to them. The questions have been designed to give them opportunities to discuss what they have found out about computer science. It is important to assess the quality of **what the student offers**.

Do not mark scripts based on the answer **you** would have written. The mark schemes have been composed to assess **quality of response** and not to identify expected items of knowledge.

Assessment Objectives

This component requires students to:

AO1: Demonstrate knowledge and understand of the key concepts and principles of computer science.

AO2: Apply knowledge and understanding of key concepts and principles of computer science.

AO3: Analyse problems in computational terms in order to develop and test programmed solutions and demonstrate an understanding of programming concepts.

The following annotation is used in the mark scheme.

;	means a single mark
//	means alternative response
/	means an alternative word or sub-phrase
A	means acceptable creditworthy answer
R	means reject answer as not creditworthy
NE	means not enough
I	means ignore
DPT	in some questions a specific error made by a student, if repeated, could result in the student failing to achieve multiple marks. The **DPT** label indicates that this mistake should result in a student not achieving only one mark, on the first occasion that the error is made. Provided that the answer remains understandable, subsequent marks should be awarded as if the error was not being repeated.

Question	Part	Marking guidance	Total marks
01	1	3;	1 AO3 = 1

<table>
<tr><th>Question</th><th>Part</th><th>Marking guidance</th><th>Total marks</th></tr>
<tr><td>01</td><td>2</td><td>
1 mark: Correct values in all cells that represent an edge (boxed below).

1 mark: Suitable indicator eg 0 or negative number in cells that do not represent an edge. R. cells empty
<table>
<tr><th></th><th></th><th colspan="5">To</th></tr>
<tr><th></th><th></th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><th rowspan="5">From</th><th>1</th><td>0</td><td>15</td><td>0</td><td>0</td><td>0</td></tr>
<tr><th>2</th><td>0</td><td>0</td><td>20</td><td>7</td><td>12</td></tr>
<tr><th>3</th><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><th>4</th><td>0</td><td>0</td><td>6</td><td>0</td><td>8</td></tr>
<tr><th>5</th><td>0</td><td>12</td><td>0</td><td>0</td><td>0</td></tr>
</table>
</td><td>2

AO3 = 2</td></tr>
</table>

Question	Part	Marking guidance	Total marks
01	3	Adjacency matrix appropriate when… • there are many edges between vertices // when graph/matrix is not sparse // when graph is dense; • when edges frequently changed; • when presence/absence of specific edges needs to be tested frequently; **A.** alternative words which describe edge, eg connection, line **Max 2**	2 AO3 = 2

Question	Part	Marking guidance	Total marks
02	1	The method by which a class operates; is hidden from other classes; **Max 2**	**2** **AO3 = 2**

Question	Part	Marking guidance	Total marks
02	2	Public means it can be accessed / seen outside of the class it is in; Protected means it can be accessed / seen in the class it is in and in any subclasses // protected means it can be accessed / seen in the class it is in and in any classes derived / inheriting from it;	**2** **AO3 = 2**

Question	Part	Marking guidance	Total marks
03	1	Static data structures have storage size determined at compile-time / before program is run / when program code is translated / before the data structure is first used // dynamic data structures can grow / shrink during execution / at run-time // static data structures have fixed (maximum) size // size of dynamic data structures can change; Static data structures can waste storage space / memory if the number of data items stored is small relative to the size of the structure // dynamic data structures only take up the amount of storage space required for the actual data; Dynamic data structures require (memory to store) pointers to the next item(s) // static data structures (typically) do not need (memory to store) pointers; Static data structures (typically) store data in consecutive memory locations // dynamic data structures (typically) do not store data in consecutive memory locations; **Max 2**	**2** **AO3 = 2**

<table>
<tr><th>Question</th><th>Part</th><th>Marking guidance</th><th>Total marks</th></tr>
<tr><td>03</td><td>2</td><td>

1 mark: After first iteration, value `JobT` and priority `3` stored in index 4.

1 mark: After second iteration, value `JobB` and priority `5` stored in index 3.

1 mark: After third iteration, value `JobX` and priority `7` stored in index 2.

1 mark: Data in index 1 and index 2 not changed in any row.

Max 3 if final content of data structure not fully correct

<table>
<tr><th>[0]</th><th>[1]</th><th>[2]</th><th>[3]</th><th>[4]</th><th>[5]</th></tr>
<tr><td>JobA</td><td>JobM</td><td>JobB</td><td>JobT</td><td>JobT</td><td></td></tr>
<tr><td>10</td><td>10</td><td>5</td><td>3</td><td>3</td><td></td></tr>
<tr><td>JobA</td><td>JobM</td><td>JobB</td><td>JobB</td><td>JobT</td><td></td></tr>
<tr><td>10</td><td>10</td><td>5</td><td>5</td><td>3</td><td></td></tr>
<tr><td>JobA</td><td>JobM</td><td>JobX</td><td>JobB</td><td>JobT</td><td></td></tr>
<tr><td>10</td><td>10</td><td>7</td><td>5</td><td>3</td><td></td></tr>
</table>

</td><td>4

AO3 = 4</td></tr>
</table>

<table>
<tr><th>Question</th><th>Part</th><th>Marking guidance</th><th>Total marks</th></tr>
<tr><td>04</td><td>1</td><td>

1. Suitable prompt displayed asking user to enter plaintext or key.
2. Plaintext or key input into appropriate variable.
3. Suitable prompts and input stored into appropriate variables for both plaintext and key.
4. Array/list of strings or alternative data structure created.
A. two-dimensional array of characters
5. Array/list or alternative data structure has dynamic size or size based on key so it always has enough rows to store the number or rails indicated by the key.
A. size is based on message length instead of key
6. Loop that iterates through each character in the plaintext.

</td><td>12

AO3 = 12</td></tr>
</table>

7. An individual character in the plaintext or key is accessed.

8. At least two characters in plaintext copied to different rails.
   I. if copied to incorrect rails

9. At least one character copied to each rail, based on key.
   I. if copied to incorrect rails

10. All characters copied to correct rails.
    R. if would only work for one specific key value

11. Content of each rail concatenated together to form ciphertext.
    I. contents of rails incorrect or concatenated in incorrect order

12. Ciphertext is output at end.
    I. ciphertext is incorrect

Max 11 if code contains any errors

Exemplar Solutions

Python

```
plaintext = input("Enter Plaintext: ")
key = int(input("Enter Key: "))
rails = ["" for railno in range(key)]
rail_num = 0
direction = "up"
for pos in range(len(plaintext)):
    rails[rail_num] += plaintext[pos]
    if direction == "up":
        rail_num +=1
        if rail_num == key:
            rail_num = key - 2
            direction = "down"
    else:
        rail_num -= 1
        if rail_num == -1:
            rail_num = 1
            direction = "up"
ciphertext = ""
for pos in range(key):
    ciphertext += rails[pos]
print("Ciphertext:", ciphertext)
```

C#

```
Console.Write("Enter Plaintext: ");
string plaintext = Console.ReadLine();
Console.Write("Enter Key: ");
int key = int.Parse(Console.ReadLine());

string[] rails = new string[key];
```

```
int railNum = 0;
bool increasing = true;

for (int pos = 0; pos < plaintext.Length; pos++)
{
    rails[railNum] += plaintext[pos];
    if (increasing)
    {
        railNum++;
        if (railNum == key)
        {
            increasing = false;
            railNum -= 2;
        }
    }
    else
    {
        railNum--;
        if (railNum == -1)
        {
            increasing = true;
            railNum += 2;
        }
    }
}

string ciphertext = "";
for (railNum = 0; railNum < key; railNum++)
{
    ciphertext+= rails[railNum];
}
Console.WriteLine("Ciphertext: " + ciphertext);
```

VB.Net

```
Console.Write("Enter Plaintext: ")
Dim PlainText = Console.ReadLine()
Console.Write("Enter Key: ")
Dim Key As Integer = Console.ReadLine()

Dim Rails(Key - 1) As String
Dim RailNum = 0
Dim RailStep = 1

For Pos = 0 To PlainText.Length - 1
    Rails(RailNum) += PlainText(Pos)
    RailNum += RailStep
    If RailNum = -1 Or RailNum = Key Then
        RailStep = -RailStep
        RailNum += RailStep * 2
    End If
Next

Dim CipherText As String = ""
For RailNum = 0 To Key - 1
    CipherText += Rails(RailNum)
Next
```

		`Console.WriteLine("Ciphertext: " & CipherText)`	

Question	Part	Marking guidance	Total marks

04	2	*Evidence must match code from* ***04.1****, including prompts matching those in code. Code for* ***04.1*** *must be sensible.*
		
		Test evidence shows:
		• `COMPUTER` input as plaintext
		• `3` input as key
		• `CUOPTRME` output as ciphertext
		
		<u>Exemplar Test Results</u>
		
		`Enter Plaintext: COMPUTER`
		`Enter Key: 3`
		`Ciphertext: CUOPTRME`
			1
			
			AO3 = 1

Question	Part	Marking guidance	Total marks
05	1	1. Data structure created that can represent node labels. **I.** incorrect labels represented 2. Data structure(s) created that can represent node labels and pointers. **I.** incorrect labels or pointers represented 3. Correct representation of labels and pointers for tree matching figure on question paper. 4. Subroutine that will visit at least one node in tree created and called. 5. Subroutine calls itself. 6. Label of current node output in subroutine. **A.** only output in some circumstances **I.** output in incorrect place **I.** output multiple times 7. Label of current node output between attempts to traverse left and right subtrees. **I.** if subtree traversal does not work **R.** output multiple times 8. Check made if current node has left child.	12 AO3 = 12

9. Call to traverse left subtree if and only if node has left child and node is passed data required to parse correct part of tree.

10. Check made if current node has right child.

11. Call to traverse right subtree if and only if node has right child and node is passed data required to parse correct part of tree.

12. Correct result of in-order traversal displayed.

Max 11 if code contains any errors

Note that only one of mark points 8 and 10 can be awarded if, when the subroutine is called, either the left or right subtree could be traversed, but not both of them.

Exemplar Solutions

Python

```
class node:
  def __init__(self, label, left_ptr, right_ptr):
    self.label = label
    self.left_ptr = left_ptr
    self.right_ptr = right_ptr

def traverse_tree(current_index, nodes):
    if nodes[current_index].left_ptr != -1:

traverse_tree(nodes[current_index].left_ptr,
nodes)
    print(nodes[current_index].label)
    if nodes[current_index].right_ptr != -1:

traverse_tree(nodes[current_index].right_ptr,
nodes)

nodes = []
nodes.append(node("D", 1, 2))
nodes.append(node("C", 3, 4))
nodes.append(node("E", -1, 5))
nodes.append(node("A", -1, -1))
nodes.append(node("B", -1, -1))
nodes.append(node("G", 6, -1))
nodes.append(node("F", -1, -1))
root_index = 0
traverse_tree(root_index, nodes)
```

C#

```
static string[] label = { "D", "C", "E", "A", "B", "G", "F" };
static int[] leftPtr = { 1, 3, -1, -1, -1, 6, -1 };
static int[] rightPtr = { 2, 4, 5, -1, -1, -1, -1 };
```

		<pre>static void TraverseTree(int currentIndex)
		{
		    if (leftPtr[currentIndex] != -1)
		        TraverseTree(leftPtr[currentIndex]);
		    Console.WriteLine(label[currentIndex]);
		    if (rightPtr[currentIndex] != -1)
		        TraverseTree(rightPtr[currentIndex]);
		}
		
		static void Main()
		{
		    int rootIndex = 0;
		    TraverseTree(rootIndex);
		    Console.ReadLine();
		}</pre>
		VB.Net
		<pre>Structure Node
		    Dim Label As String
		    Dim LeftPtr As String
		    Dim RightPtr As String
		End Structure
		
		Sub TraverseTree(CurrentIndex As Integer, Tree() As Node)
		    If Tree(CurrentIndex).LeftPtr <> -1 Then
		        TraverseTree(Tree(CurrentIndex).LeftPtr, Tree)
		    End If
		    Console.WriteLine(Tree(CurrentIndex).Label)
		    If Tree(CurrentIndex).RightPtr <> -1 Then
		        TraverseTree(Tree(CurrentIndex).RightPtr, Tree)
		    End If
		End Sub
		
		Sub Main()
		    Dim Tree(6) As Node
		    Tree(0).Label = "D"
		    Tree(0).LeftPtr = 1
		    Tree(0).RightPtr = 2
		    Tree(1).Label = "C"
		    Tree(1).LeftPtr = 3
		    Tree(1).RightPtr = 4
		    Tree(2).Label = "E"
		    Tree(2).LeftPtr = -1
		    Tree(2).RightPtr = 5
		    Tree(3).Label = "A"
		    Tree(3).LeftPtr = -1
		    Tree(3).RightPtr = -1
		    Tree(4).Label = "B"
		    Tree(4).LeftPtr = -1
		    Tree(4).RightPtr = -1
		    Tree(5).Label = "G"
		    Tree(5).LeftPtr = 6
		    Tree(5).RightPtr = -1
		    Tree(6).Label = "F"
		    Tree(6).LeftPtr = -1
		    Tree(6).RightPtr = -1</pre>	

		`    TraverseTree(0, Tree)` `End Sub`	

Question	Part	Marking guidance	Total marks

05	**2**	*Evidence must match code from **05.1**, including prompts matching those in code. Code for **05.1** must be sensible.*
		
		Test evidence shows correct output of in-order traversal: `ABCDEFG`.
		
		<u>Exemplar Test Results</u>
		
		`A`
		`C`
		`B`
		`D`
		`E`
		`F`
		`G`
			1
			
			AO3 = 1

Question	Part	Marking guidance	Total marks
06	**1**	**For data structure:** 1. Data structure created that can represent one student. 2. Data structure created that can represent 100 values. **I.** values are not students **R.** list/dynamically sized data structure created unless there is also code to expand the size of the data structure if necessary when a new student is added **For hash value calculation:** 3. A calculation is performed to work out a hash value. **I.** incorrect calculation 4. Hash calculation includes either the position in the alphabet of the first letter in the `StudentID` or the last two digits in the `StudentID`. 5. Hash value correctly calculated. **For `AddStudent` method:** 6. `AddStudent` method created and takes `StudentID`, `Forename` and `Surname` as parameters. 7. Method stores student details into data structure.	**21** **AO3 = 21**

		8. Method stores student details at row in data structure indicated by hash value. **I.** incorrect hash value 9. If collision occurs, error message displayed and new data not stored. **For `LookupStudent` method:** 10. `LookupStudent` method created and takes `StudentID` as parameter. 11. If record for student stored, correct details of student are displayed. **I.** StudentID is not displayed 12. Location of student record calculated using hash function. **I.** incorrect hash value 13. Error message output if student details not stored. **For `DeleteStudent` method:** 14. `DeleteStudent` method created and takes `StudentID` as parameter. 15. If record for student stored, location in hash table marked so that it can be reused. **For OOP program structure:** 16. Hash table created as a class. 17. Instance of hash table class created. 18. Class properties are all private – must be at least one valid data structure. 19. Methods that must be called from outside class are public, any methods only used internally are private – must be at least one required method. All methods that the class uses are declared within the class. **For data stored in data structure:** 20. At least two of the additions/deletions/lookups of the hash table are made by calling the appropriate methods. **I.** if code called does not work 21. All five of the additions/deletions/lookups of the hash table are made in the correct order by calling the appropriate methods. **I.** if code called does not work The required additions/deletions/lookups are:	

- Add student DH409 Peter Smith
- Add student FP789 Ibrahim Saleem
- Lookup student DH409
- Delete student DH409
- Lookup student DH409 a second time

Max 20 if code contains any errors

Exemplar Solutions

Python

```
class student:
    def __init__(self):
        self.student_ID = "X"
        self.forename = "X"
        self.surname = "X"

class hash_table:
    def __init__(self):
        self.__table = [student() for row in
range(100)]

    def __calculate_hash(self, student_ID):
        hash = ((ord(student_ID[0]) - 64) * 10 +
int(student_ID[3:5])) % 100
        return hash

    def add_student(self, student_ID, forename,
surname):
        hash = self.__calculate_hash(student_ID)
        if self.__table[hash].student_ID != "X":
            print("Collision")
        else:
            self.__table[hash].student_ID =
student_ID
            self.__table[hash].forename = forename
            self.__table[hash].surname = surname

    def lookup_student(self, student_ID):
        hash = self.__calculate_hash(student_ID)
        if self.__table[hash].student_ID == "X":
            print("Student does not exist")
        else:
            print("Student ID:",
self.__table[hash].student_ID)
            print("Forename:",
self.__table[hash].forename)
            print("Surname:",
self.__table[hash].surname)

    def delete_student(self, student_ID):
        hash = self.__calculate_hash(student_ID)
```

		<pre> self.__table[hash].student_ID = "X"
		        self.__table[hash].forename = "X"
		        self.__table[hash].surname = "X"
		
		student_hash_table = hash_table()
		student_hash_table.add_student("DH409","Peter","Sm
		ith")
		student_hash_table.add_student("FP789", "Ibrahim",
		"Saleem")
		student_hash_table.lookup_student("DH409")
		student_hash_table.delete_student("DH409")
		student_hash_table.lookup_student("DH409")</pre>
		C#
		<pre>class HashTable
		{
		    struct StudentRecord
		    {
		        public string studentID;
		        public string forename;
		        public string surname;
		    }
		
		    private StudentRecord[] table = new StudentRecord[100];
		
		    public HashTable()
		    {
		        for (int row = 0; row < 100; row++)
		        {
		            table[row].studentID = "X";
		            table[row].forename = "X";
		            table[row].surname = "X";
		        }
		    }
		
		    private int CalculateHash(string studentID)
		    {
		        int hash = (((int)studentID[0] - 64) * 10 +
		Convert.ToInt32(studentID.Substring(3))) % 100;
		        return hash;
		    }
		
		    public void AddStudent(string studentID, string forename,
		string surname)
		    {
		        int row = CalculateHash(studentID);
		        if (table[row].surname == "X")
		        {
		            table[row].studentID = studentID;
		            table[row].forename = forename;
		            table[row].surname = surname;
		        }
		        else
		        {
		            Console.WriteLine("Collision");
		        }</pre>	

		<pre> }
		
		    public void LookupStudent(string studentID)
		    {
		        int row = CalculateHash(studentID);
		        if (table[row].surname == "X")
		        {
		            Console.WriteLine("Student does not exist");
		        }
		        else
		        {
		            Console.WriteLine("Student ID: " +
		table[row].studentID);
		            Console.WriteLine("Forename: " +
		table[row].forename);
		            Console.WriteLine("Surname: " +
		table[row].surname);
		        }
		    }
		
		    public void DeleteStudent(string studentID)
		    {
		        int row = CalculateHash(studentID);
		        table[row].studentID = "X";
		        table[row].forename = "X";
		        table[row].surname = "X";
		    }
		}
		
		class Program
		{
		    static void Main()
		    {
		        HashTable studentHashTable = new HashTable();
		        studentHashTable.AddStudent("DH409", "Peter", "Smith");
		        studentHashTable.AddStudent("FP789", "Ibrahim",
		"Saleem");
		        studentHashTable.LookupStudent("DH409");
		        studentHashTable.DeleteStudent("DH409");
		        studentHashTable.LookupStudent("DH409");
		        Console.ReadLine();
		    }
		}</pre>
		VB.Net
		<pre>Class HashTable
		    Private StudentRecord(99, 2) As String
		    Sub New()
		        For Row = 0 To 99
		            For Col = 0 To 2
		                StudentRecord(Row, Col) = ""
		            Next
		        Next
		    End Sub
		
		    Private Function CalculateHash(StudentID As String) As
		Integer</pre>	

| | | ```
        Dim Hash = ((Convert.ToByte(StudentID(0)) - 64) * 10 +
Convert.ToInt32(StudentID.Substring(3))) Mod 100
        Return Hash
    End Function

    Public Sub AddStudent(StudentID As String, Forename As
String, Surname As String)
        Dim Row = CalculateHash(StudentID)
        If StudentRecord(Row, 0) = "" Then
            StudentRecord(Row, 0) = StudentID
            StudentRecord(Row, 1) = Forename
            StudentRecord(Row, 2) = Surname
        Else
            Console.WriteLine("Collision")
        End If
    End Sub

    Public Sub LookUpStudent(StudentID As String)
        Dim Row = CalculateHash(StudentID)
        If StudentRecord(Row, 0) = "" Then
            Console.WriteLine("Student does not exist")
        Else
            Console.WriteLine("Student ID: " &
StudentRecord(Row, 0))
            Console.WriteLine("Forename: " & StudentRecord(Row,
1))
            Console.WriteLine("Surname: " & StudentRecord(Row,
2))
        End If
    End Sub

    Public Sub DeleteStudent(StudentID As String)
        Dim Row = CalculateHash(StudentID)
        StudentRecord(Row, 0) = ""
        StudentRecord(Row, 1) = ""
        StudentRecord(Row, 2) = ""
    End Sub

End Class

Sub Main()
    Dim StudentHashTable As HashTable = New HashTable()
    StudentHashTable.AddStudent("DH409", "Peter", "Smith")
    StudentHashTable.AddStudent("FP789", "Ibrahim", "Saleem")
    StudentHashTable.LookUpStudent("DH409")
    StudentHashTable.DeleteStudent("DH409")
    StudentHashTable.LookUpStudent("DH409")
    Console.ReadLine()
End Sub
``` | |
```

<table>
<tr><th>Question</th><th>Part</th><th>Marking guidance</th><th>Total marks</th></tr>
<tr><td>06</td><td>2</td><td>Evidence must match code from 06.1, including prompts matching those in code. Code for 06.1 must be sensible.

Test evidence shows:
error message indicating collision has occurred
details of student Peter Smith displayed
error message indicating student does not exist.
<u>Exemplar Test Results</u>

<code>Collision
Student ID: DH409
Forename: Peter
Surname: Smith
Student does not exist</code></td><td>1

AO3 = 1</td></tr>
</table>

<table>
<tr><th>Question</th><th>Part</th><th>Marking guidance</th><th>Total marks</th></tr>
<tr><td>07</td><td>1</td><td>For data representation

1. Suitable data structure created to store representation of board.

2. Variable created to represent which player’s turn it is.

For column input:

3. Suitable prompt asking user to input column to drop counter into displayed and input assigned to appropriate variable.

4. Check if column valid/invalid.
NE. only one boundary checked

5. Check if column is already full.

6. Error message displayed or re-entry required if either column invalid or column already full.
I. check is only partially correct eg only checks if too high not too low

7. Error message displayed and re-entry required if either column invalid or column already full.

For counter placement:

8. Counter stored in correct column in data structure, based on user input.</td><td>26

AO3 = 26</td></tr>
</table>

		9. Attempt to identify correct row to store counter in, eg using loop or storing column heights. 10. Counter stored in correct row in data structure, based on user input and counters already placed in column. **R.** counter placed in more than one row **For board display:** 11. Loop iterates through rows. 12. Loop iterates through columns. 13. Correct display of board, based on contents of data structure. **I.** contents of data structure incorrect **For winner identification and game termination:** 14. Check for row of three in horizontal direction in at least one location on board. 15. Any valid winning row correctly identified. **R.** check would sometimes go outside of array bounds 16. Check for column of three in horizontal direction in at least one location on board. 17. Any valid winning column correctly identified. **R.** check would sometimes go outside of array bounds 18. Suitable message output when program identifies winner. **I.** if the conditions used to identify a winner are not correct **For basic gameplay:** 19. Loop used to give repeated turns at playing game. 20. For each turn, the board is displayed, the user is able to select the column for a counter and the counter is placed in the column. 21. The player dropping a counter alternates between turns. 22. Game terminates when there is a winner OR game terminates when board is full. **For program structure:** 23. At least one user-defined subroutine created and called, which has an appropriate meaningful name. 24. Appropriate overall division of program into subroutines. **Note:** Must be at least three programmer-created subroutines 25. No repetition of code to achieve the same purpose in more than one place. For example, code to display board is not duplicated,	

code to play game and test for a winner is not duplicated for both players.
Note: Some attempt must have been made to write code for both players to award this mark

26. No use of global variables, all values passed between subroutines using parameters and return values.

Max 25 if code contains any errors

<u>Exemplar Solutions</u>

Python

```
def display_board(board):
    for row in range (4, -1, -1):
        for col in range(6):
            print(board[row][col], end="")
        print()

def enter_column(rows_used):
    repeat = True
    while repeat:
        col = int(input("Select column: "))
        if col < 0 or col > 5:
            print("Invalid column")
        elif rows_used[col] == 5:
            •  print("Column full")
        else:
            repeat = False
    return col

def check_winner(board, row, col, player_one_turn):
    winner = False
    if player_one_turn:
        symbol = '1'
    else:
        symbol = '2'
    try:
        if board[row][col] == symbol and board[row + 
1][col] == symbol and board[row + 2][col] == symbol:
            winner = True
    except:
       pass
    try:
        if board[row - 1][col] == symbol and 
board[row][col] == symbol and board[row + 1][col] == 
symbol:
            winner = True
    except:
        pass
    try:
        if board[row - 2][col] == symbol and board[row 
- 1][col] == symbol and board[row][col] == symbol:
```

```
            winner = True
    except:
        pass
    try:
        if board[row][col] == symbol and board[row][col
+ 1] == symbol and board[row][col + 2] == symbol:
            winner = True
    except:
       pass
    try:
        if board[row][col - 1] == symbol and
board[row][col] == symbol and board[row][col + 1] ==
symbol:
            winner = True
    except:
       pass
    try:
        if board[row][col - 2] == symbol and
board[row][col - 1] == symbol and board[row][col] ==
symbol:
            winner = True
    except:
       pass
    return winner

board = [[' ' for col in range(6)] for row in range(5)]
rows_used = [0 for col in range(6)]
game_over = False
player_one_turn = True
while game_over == False:
    display_board(board)
    col = enter_column(rows_used)
    row = rows_used[col]
    if player_one_turn:
        board[row][col] = '1'
    else:
        board[row][col] = '2'
    rows_used[col] += 1
    winner = check_winner(board, row, col,
player_one_turn)
    if winner:
        display_board(board)
        print("Game won")
        game_over = True
    if sum(rows_used) == 6 * 5:
        display_board(board)
        game_over = True
    player_one_turn = not player_one_turn
```

C#

```
static void DisplayBoard(int[,] board)
{
    for (int row = 4; row >= 0; row--)
```

		<pre> {
		        for (int col = 0; col < 6; col++)
		        {
		            if (board[row, col] == 0) Console.Write(" ");
		            else Console.Write(board[row, col] + " ");
		        }
		        Console.WriteLine();
		    }
		}
		
		static int FindFreeRow(int col, int[,] board)
		{
		    int row = 0;
		    if (board[4, col] > 0) return -1;
		    else
		    {
		         while (board[row, col] != 0) row++;
		    }
		    return row;
		}
		
		static int InputColumn(int[,] board)
		{
		    int col;
		    int freeRow;
		    bool repeat;
		    do
		    {
		        repeat = false;
		        Console.Write("Select column: ");
		        col = int.Parse(Console.ReadLine());
		        if (col < 0 || col > 5)
		        {
		            Console.WriteLine("Invalid column");
		            repeat = true;
		        }
		        else
		        {
		            freeRow = FindFreeRow(col, board);
		            if (freeRow == -1)
		            {
		                Console.WriteLine("Column full");
		                repeat = true;
		            }
		            Console.WriteLine(freeRow);
		        }
		    } while (repeat);
		    return col;
		}
		
		static bool CheckWinner(int[,] board)
		{
		    bool win = false;
		    for (int row = 0; row <= 4; row++)
		    {
		        for (int col = 0; col <= 3; col++)
		        {</pre>	

		<pre> if (board[row, col] == board[row, col + 1] &&
		board[row, col] == board[row, col + 2] && board[row, col] > 0)
		            {
		                win = true;
		            }
		        }
		    }
		    for (int row = 0; row <= 2; row++)
		    {
		        for (int col = 0; col <= 5; col++)
		        {
		            if (board[row, col] == board[row + 1, col] &&
		board[row, col] == board[row + 2, col] && board[row, col] > 0)
		            {
		                win = true;
		            }
		        }
		    }
		    if (win) Console.WriteLine("Game has been won");
		    return win;
		}
		
		static void Main()
		{
		    int[,] board = new int[5, 6];
		    int player = 1;
		    int turns = 0;
		    do
		    {
		        DisplayBoard(board);
		        int col = InputColumn(board);
		        int row = FindFreeRow(col, board);
		        board[row, col] = player;
		        player = 3 - player;
		        turns++;
		    } while (!CheckWinner(board) && turns < 5 * 6);
		    DisplayBoard(board);
		    Console.ReadLine();
		}</pre>
		VB.Net
		<pre>Sub DisplayBoard(Board(,) As Char)
		    For Row = 4 To 0 Step -1
		        For Col = 0 To 5
		            Console.Write(Board(Row, Col) & " ")
		        Next
		        Console.WriteLine()
		    Next
		End Sub
		
		Function FindFreeRow(Col As Integer, Board(,) As Char) As
		Integer
		    Dim Row As Integer = -1
		    Do
		        Row += 1
		    Loop Until Row = 4 Or Board(Row, Col) = " "</pre>	

```
    If Board(Row, Col) = " " Then
        Return Row
    Else
        Return 99
    End If
End Function

Function InputColumn() As Integer
    Dim Col As Integer
    Do
        Console.Write("Select column: ")
        Col = Console.ReadLine()
        If Col < 0 Or Col > 5 Then
            Console.WriteLine("Invalid Column")
        End If
    Loop Until Col >= 0 And Col <= 5
    Return Col
End Function

Function CheckWinner(Board(,) As Char) As Boolean
    Dim Win As Boolean = False
    For Row = 0 To 4
        For Col = 0 To 3
            If Board(Row, Col) = Board(Row, Col + 1) And
Board(Row, Col) = Board(Row, Col + 2) And Board(Row, Col) <> "
" Then
                Win = True
            End If
        Next
    Next
    For Row = 0 To 2
        For Col = 0 To 5
            If Board(Row, Col) = Board(Row + 1, Col) And
Board(Row, Col) = Board(Row + 2, Col) And Board(Row, Col) <> "
" Then
                Win = True
            End If
        Next
    Next
    If Win Then Console.WriteLine("Game has been won")
    Return Win
End Function

Sub ResetBoard(Board(,) As Char)
    For Row = 0 To 4
        For Col = 0 To 5
            Board(Row, Col) = " "
        Next
    Next
End Sub

Sub Main()
    Dim Board(4, 5) As Char
    ResetBoard(Board)
    Dim Player As Char = "1"
    Dim Turns As Integer = 0
    Do
        DisplayBoard(Board)
```

		<pre> Dim Col As Integer
		        Dim Row As Integer
		        Do
		            Col = InputColumn()
		            Row = FindFreeRow(Col, Board)
		            If Row = 99 Then
		                Console.WriteLine("Column full")
		            End If
		        Loop Until Row <> 99
		        Board(Row, Col) = Player
		        If Player = "1" Then
		            Player = "2"
		        Else
		            Player = "1"
		        End If
		        Turns += 1
		    Loop Until CheckWinner(Board) Or Turns = 5 * 6
		    DisplayBoard(Board)
		    Console.ReadLine()
		End Sub</pre>	

Question	Part	Marking guidance	Total marks

07	2	*Evidence must match code from **07.1**, including prompts matching those in code. Code for **07.1** must be sensible.*
		
		Test evidence shows game played until there is a winner.
		
		<u>Exemplar Test Results</u>
		
		`Select column: 0`
		
		`1`
		`Select column: 1`
		
		`12`
		`Select column: 0`
		
		`1`
		`12`
		`Select column: 1`
		
		`12`
		`12`
		`Select column: 0`
		
		`1`
		`12`
		`12`
		`Game won`
			1
			
			AO3 = 1