AQA

A-level
COMPUTER SCIENCE
7517/1

Paper 1

Mark scheme

June 2021

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 aqa.org.uk

Level of response marking instructions

Level of response mark schemes are broken down into levels, each of which has a descriptor. The descriptor for the level shows the average performance for the level. There are marks in each level.

Before you apply the mark scheme to a student's answer read through the answer and annotate it (as instructed) to show the qualities that are being looked for. You can then apply the mark scheme.

Step 1 Determine a level

Start at the lowest level of the mark scheme and use it as a ladder to see whether the answer meets the descriptor for that level. The descriptor for the level indicates the different qualities that might be seen in the student's answer for that level. If it meets the lowest level then go to the next one and decide if it meets this level, and so on, until you have a match between the level descriptor and the answer. With practice and familiarity you will find that for better answers you will be able to quickly skip through the lower levels of the mark scheme.

When assigning a level you should look at the overall quality of the answer and not look to pick holes in small and specific parts of the answer where the student has not performed quite as well as the rest. If the answer covers different aspects of different levels of the mark scheme you should use a best fit approach for defining the level and then use the variability of the response to help decide the mark within the level, ie if the response is predominantly level 3 with a small amount of level 4 material it would be placed in level 3 but be awarded a mark near the top of the level because of the level 4 content.

Step 2 Determine a mark

Once you have assigned a level you need to decide on the mark. The descriptors on how to allocate marks can help with this. The exemplar materials used during standardisation will help. There will be an answer in the standardising materials which will correspond with each level of the mark scheme. This answer will have been awarded a mark by the Lead Examiner. You can compare the student's answer with the example to determine if it is the same standard, better or worse than the example. You can then use this to allocate a mark for the answer based on the Lead Examiner's mark on the example.

You may well need to read back through the answer as you apply the mark scheme to clarify points and assure yourself that the level and the mark are appropriate.

Indicative content in the mark scheme is provided as a guide for examiners. It is not intended to be exhaustive and you must credit other valid points. Students do not have to cover all of the points mentioned in the Indicative content to reach the highest level of the mark scheme.

An answer which contains nothing of relevance to the question must be awarded no marks.

A-level Computer Science

Paper 1 (7517/1) – applicable to all programming languages A, B, C, D and E

June 2021

The following annotation is used in the mark scheme:

; - means a single mark
// - means an alternative response
/ - means an alternative word or sub-phrase
A. - means an acceptable creditworthy answer
R. - means reject answer as not creditworthy
NE. - means not enough
I. - means ignore
DPT. - means "Don't penalise twice". In some questions a specific error made by a candidate, if repeated, could result in the loss of more than one mark. The **DPT** label indicates that this mistake should only result in a candidate losing 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.

Examiners are required to assign each of the candidate's responses to the most appropriate level according to **its overall quality**, and then allocate a single mark within the level. When deciding upon a mark in a level examiners should bear in mind the relative weightings of the assessment objectives

eg

In question **07.1**, the marks available for the AO3 elements are as follows:

AO3 (design)	4 marks
AO3 (programming)	8 marks

Where a candidate's answer only reflects one element of the AO, the maximum mark they can receive will be restricted accordingly.

Question			Marks
01		**All marks AO2 (apply)** 		[0]	[1]	[2]	[3]	[4]	[5]
---	---	---	---	---	---	---
	3	5	8	1	6	4
First pass	3	5	1	6	4	8
Second pass	3	1	5	4	6	8
Third pass	1	3	4	5	6	8 | **Mark as follows:** **1 mark:** 1st row correct **1 mark:** 2nd row correct **1 mark:** 3rd row correct **Alternative answer** |	[0]	[1]	[2]	[3]	[4]	[5]
---	---	---	---	---	---	---
	3	5	8	1	6	4
First pass	1	3	5	8	4	6
Second pass	1	3	5	4	8	6
Third pass	1	3	4	5	6	8		**3**

Question			Marks
02	1	**All marks AO1 (knowledge)** Rooted (tree); Where each node has at most two child nodes; **R.** each node has two child nodes	2
02	2	**All marks AO2 (apply)** `EIHCYBQ`;; If not fully correct award a maximum of 1 mark for any of the following: • Having `E` followed by `I` then `H` • Having `Y` followed by `B` then `Q` • Having `C` as the 4th output	2

Question			Marks
02	3	**All marks AO2 (apply)**	7

		Stack				
Current	**Pos**	**[0]**	**[1]**	**[2]**	**[3]**	**OUTPUT**
0	0	0				
0	-1					C
	0	4				
	1		1			
1	0					I
	1		3			
	2			2		
2	1					E
3	0					H
4	-1					B
	0	6				
	1		5			
5	0					Y
6	-1					Q

Mark as follows:

1. `Stack[0]` set to `0`, `Pos` set to `0` and `Current` set to `0`
2. `Current` set to `0`, `Pos` set to `-1` and output of `C`
3. `Stack[0]` set to `4` and `Pos` set to `0`
4. `Stack[1]` set to `1`, then `3` and then `5` with no other values after being set to `5`
5. `Stack[2]` set to `2` with no other values after this, `Stack[0]` having a 3rd value of `6` with no other values after this and `Stack[3]` column not used;
6. `Pos` column correct from 4th value (`1`) onwards and `Current` column set to the value `1`, then `2`, `3`, `4`, `5`, `6` with no further values after being set to `6`
7. Correct order in output column for 2nd value onwards (`I`, `E`, `H`, `B`, `Y`, `Q`)

Max 6 if any errors

Question			Marks
02	4	**Mark is for AO1 (understanding)** A subroutine that calls itself;	1
02	5	**Mark is for AO1 (understanding)** The circumstance(s) when a recursive subroutine does not call itself;	1
02	6	**All marks AO1 (knowledge)** local variables; return address; parameters; register values; **A.** example of register that would be in stack frame **Max 2**	2

Question			Marks
03		**One mark for AO1 (knowledge) and two marks for AO1 (understanding)** **AO1 knowledge** Breaking a problem into smaller sub-problems; **AO1 understanding** Each of which solves an identifiable task; Each of which might be further subdivided;	3

Question			Marks
04		**All marks AO1 (understanding)** Hash algorithm applied; to key value; **NE.** to data/item result is location in table where the record should be stored; if location is not empty; then use next free location; **A.** description of any feasible collision resolution method	5

Question			Marks
05	1	**All marks AO1 (understanding)** Simpler for a machine/computer to evaluate (**A.** easier **R.** to understand); Simpler to code algorithm; Do not need brackets (to show correct order of evaluation/calculation); **A.** RPN expressions cannot be ambiguous as BOD Operators appear in the order required for computation; No need for order of precedence of operators; No need to backtrack when evaluating; **Max 2**	**2**
05	2	**All marks AO1 (understanding)** (Starting at LHS of expression) push values/operands on to stack; **R.** if operators are also pushed onto stack, unless they are immediately popped off the stack Each time operator reached pop top two values off stack (and apply operator to them) // Each time operator reached pop required number of values off stack (and apply operator to them); Push result (of applying operator) to stack; When end of expression is reached the top item of the stack is the result // when end of expression is reached pop one value off the stack; **Max 3** if any errors **Max 3** if more than one stack used **Note for examiners:** award 0 marks if description is not about a stack / LIFO structure even if the word “stack” has been used	**4**

<table>
<tr><th colspan="2">Question</th><th></th><th>Marks</th></tr>
<tr><td>06</td><td>1</td><td>All marks AO2 (analyse)

<table>
<tr><th>Current state</th><th>Input</th><th>New state</th></tr>
<tr><td>S2</td><td>a</td><td>S5</td></tr>
<tr><td>S2</td><td>b</td><td>S4</td></tr>
<tr><td>S0</td><td>b</td><td>S2</td></tr>
<tr><td>S5</td><td>b</td><td>S2</td></tr>
</table>

Mark as follows:

• 1 mark: rows with current state of S2 are correct
• 1 mark: rows with new state of S2 are correct

I. order of rows</td><td>2</td></tr>
<tr><td>06</td><td>2</td><td>All marks AO2 (analyse)

<code>a(ba)*|b(ab)*</code>
//
<code>(a(ba)*)|(b(ab)*)</code>
//
<code>b(ab)*| a(ba)*</code>
//
<code>(b(ab)*)| (a(ba)*)</code>
//
<code>a|b|b(ab)+| a(ba)+</code>;;;

Max 2 if not fully correct

If answer is not completely correct award marks for the following:
• Expression uses two <code>*</code> metacharacters and a <code>|</code> metacharacter;
• <code>(ba)*</code> and <code>(ab)*</code> in expression; R. <code>ba*</code> R. <code>ab*</code>
• Expression will match with single <code>a</code> and a single <code>b</code>
• <code>(ba)+</code> and <code>(ab)+</code> in expression; R. <code>ba+</code> R. <code>ab+</code></td><td>3</td></tr>
</table>

<table>
<tr><th colspan="2">Question</th><th></th><th>Marks</th></tr>
<tr><td>07</td><td>1</td><td>
4 marks for AO3 (design) and 8 marks for AO3 (programming)

<u>Mark Scheme</u>
<table>
<tr><th>Level</th><th>Description</th><th>Mark Range</th></tr>
<tr><td>4</td><td>A line of reasoning has been followed to arrive at a logically structured working or almost fully working programmed solution that meets most of the requirements. All of the appropriate design decisions have been taken. To award 12 marks, all of the requirements must be met.</td><td>10–12</td></tr>
<tr><td>3</td><td>There is evidence that a line of reasoning has been followed to produce a logically structured program. The program displays relevant prompts, inputs the required number, has at least one iterative structure and one selection structure and uses appropriate variables to store most of the needed data. An attempt has been made to determine if a number is a Harshad number, although this may not work correctly under all circumstances. The solution demonstrates good design work as most of the correct design decisions have been made.</td><td>7–9</td></tr>
<tr><td>2</td><td>A program has been written and some appropriate, syntactically correct programming language statements have been written. There is evidence that a line of reasoning has been partially followed as although the program may not have the required functionality, it can be seen that the response contains some of the statements that would be needed in a working solution. There is evidence of some appropriate design work as the response recognises at least one appropriate technique that could be used by a working solution, regardless of whether this has been implemented correctly.</td><td>4–6</td></tr>
<tr><td>1</td><td>A program has been written and a few appropriate programming language statements have been written but there is no evidence that a line of reasoning has been followed to arrive at a working solution. The statements written may or may not be syntactically correct. It is unlikely that any of the key design elements of the task have been recognised.</td><td>1–3</td></tr>
</table>
</td><td>12</td></tr>
</table>

Guidance

Evidence of AO3 design – 4 points:

Evidence of design to look for in responses:

1. Identifying that integer division is needed when calculating the sum of the digits // identifying that a character in string needs to be converted to a number data type when calculating the sum of the digits
2. Identifying that a loop is needed that repeats a number of times determined by the number entered by the user // identifying that a loop is needed that repeats until the *n*th Harshad number is found
3. Identifying that nested iteration is needed
4. Selection structure that compares sum of digits (**I.** incorrectly calculated) with a number

Note that AO3 (design) points are for selecting appropriate techniques to use to solve the problem, so should be credited whether the syntax of programming language statements is correct or not and regardless of whether the solution works.

Evidence for AO3 programming – 8 points:

Evidence of programming to look for in response:

5. Suitable prompt asking user to enter a number followed by user input being assigned to appropriate variable
6. Iterative structure that repeats a number of times sufficient to find all the digits of a number
7. Calculates the sum of all the digits of a number
8. Calculates the remainder from dividing a number by its sum of digits **A.** incorrect calculation for sum of digits
9. Resets the variable used to store the sum of digits to 0 in an appropriate place
10. Program works correctly for the first nine Harshad numbers (1 to 9)
11. Program will display 10/12/18 if the user enters the number 10/11/12
12. Program displays the correct value for the *n*th Harshad number under all circumstances **I.** displaying Harshad numbers that appear before the *n*th Harshad number

Alternative mark scheme
This mark scheme is to be used if solution uses a recursive subroutine to calculate the sum of the digits.

3. Identifying that a recursive subroutine is needed to calculate the sum of the digits.
6. Recursive subroutine has an appropriate base case.
9. Sets the variable used to store the sum of digits to the result returned by the call to the recursive subroutine in an appropriate place.

Max 11 if any errors.

07	**2**	**Mark is for AO3 (evaluate)** ****** SCREEN CAPTURE ****** *Must match code from **07.1**, including prompts on screen capture matching those in code.* *Code for **07.1** must be sensible.* Screen capture showing the number 600 being entered and then a message displayed saying 3102  	**1**

Question			Marks
08	1	**Mark is for AO2 (analyse)** If they have the same value as each other for one of their coordinates;	1
08	2	**Marks are for AO2 (analyse)** (The result of) the integer division by 2; on the x coordinate;	2
08	3	**Marks are for AO2 (analyse)** Change the 3 in the selection structure to a 7; Change the 2 (3 in Python) in the for loop to a 6 (7 in Python); **If answer is incorrect award 1 mark** for changing the 3 to a 5 and the 2 to a 4 **Alternative answer** The number of elements in the items list now needs to be seven instead of three; and would need to put three integers in the list for each tile;	2

Question			Marks
09	1	**Mark is for AO2 (analyse)** The classes that inherit from `Piece` would not be able to use it; **A.** answers that use a specific class that would not be able to use it (`Baron`, `LESS`, `PBDS`).	1
09	2	**Marks are for AO2 (analyse)** When both players' barons are destroyed in the same turn; and it is not player two's turn // and it is player one's turn;	2
09	3	**Mark is for AO1 (understanding)** A method shared (up and down the inheritance hierarchy chain) but with each class / method implementing it differently // A single interface is provided to entities/objects of different classes / types // Objects of different classes / types respond differently to the use of a common interface / the same usage // Allowing different classes to be used with the same interface // The ability to process objects differently depending on their class / type;	1

Question			Marks
10	1	**Mark is for AO2 (analyse)** The structure of the data in the file does not match the expected format; **A.** by example, eg there are not five items in the first line in the file File is not a text file; **A.** any reasonable example of a file error that could cause an exception apart from file not existing The program tries to convert a non-integer (**A.** non-numeric) (**A.** string or other example of an invalid data type) value to an integer; Program tries to store a value which is too large to be an integer as an integer; **Max 1**	1
10	2	**Mark is for AO2 (analyse)** `CheckMoveCommandFormat`; `CheckStandardCommandFormat`; `CheckUpgradeCommandFormat`; `hasMethod`; (Java only) `readLine`; (Java only) `executeCommandInTile`; (Java only) **Max 1** **R.** if spelt incorrectly **R.** if any additional code **I.** case and spacing	1

Question			Marks
11		**All marks are for AO2 (analyse)** It gets the largest of; the differences between the x coordinates, the y coordinates and the z coordinates (of the two tiles);	2

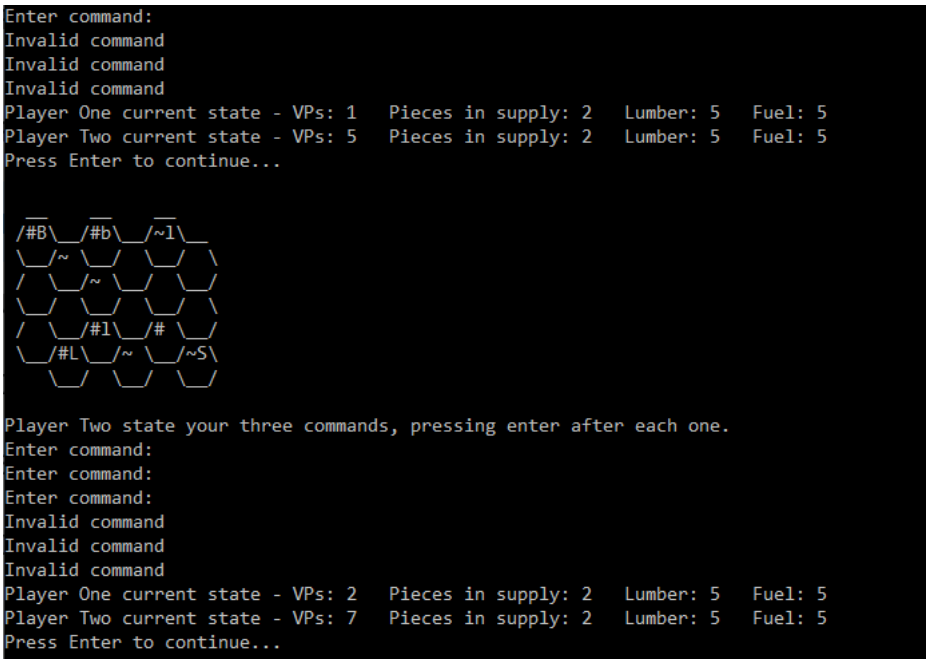
<table>
<tr><th colspan="2">Question</th><th></th><th>Marks</th></tr>
<tr><td>12</td><td>1</td><td>All marks for AO3 (programming)

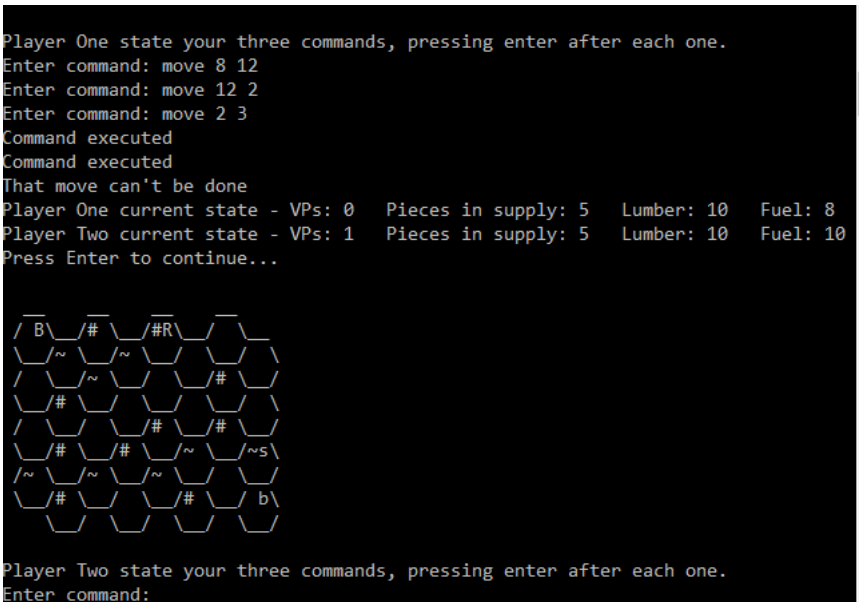
1. Correctly checks if a piece belongs to a player;
2. Correctly checks if a piece is a LESS piece;
3. Correct logic for selection structure for a player's LESS piece and one added to that player's victory points if a piece is a LESS piece belonging to that player;
4. Mark points 1 to 3 done for other player;
5. Only adds victory points for LESS pieces if they have not been destroyed;

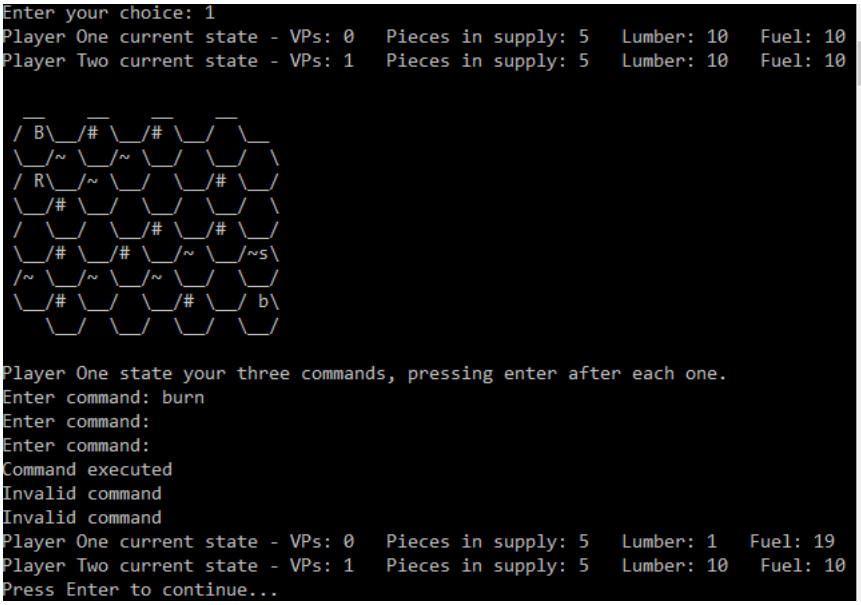
Max 4 if code contains errors</td><td>5</td></tr>
<tr><td>12</td><td>2</td><td>Mark is for AO3 (evaluate)

**** SCREEN CAPTURE ****
Must match code from 12.1.
Code for 12.1 must be sensible.

Screen captures showing the correct VP totals for both players (2 for player one and 7 for player two);

 
</td><td>1</td></tr>
</table>

Question			Marks
13	1	**All marks for AO3 (programming)** 1. Creating a new class called `RangerPiece`; **R.** other names for class **I.** case and minor typos 2. New class inherits from `Piece` and has a constuctor that overrides base class constructor with call made to base class constructor; **R.** if incorrect parameters 3. Constructor sets `PieceType` to "`R`"; **R.** if before call to base class constructor **R.** "`r`" 4. Subroutine called `CheckMoveIsValid` created that overrides base class method and correct code for normal move **R.** if incorrect parameters 5. Selection structure with correct conditions that allow move from forest terrain to forest terrain; 6. Correct fuel cost returned for all moves (forest to forest, distance of one, illegal move, distance of one with peat bog as start terrain, distance of one with peat bog as end terrain); The following relates to the `AddPiece` subroutine: 7. Selection structure with correct condition in appropriate place in code which results in call to constructor for new class; **Max 6** if code contains errors	7
13	2	**Mark is for AO3 (evaluate)** ****** SCREEN CAPTURE ****** *Must match code from **13.1**.* *Code for **13.1** must be sensible.* Screen capture(s) showing that two commands were executed and third command wasn't followed by grid with `R` piece in 3rd cell on top row;  	1

Question			Marks
14	1	**All marks for AO3 (programming)** Marks for changes to the `ExecuteCommand` method: 1. selection structure with correct condition for `burn` command; 2. selection structures with correct condition to check that there is lumber in the player's supply; 3. returns correct string (**A.** minor typos, **I.** case) if player has no lumber; 4. generates a random integer; 5. random integer generated is in correct range; 6. reduces lumber by correct amount; 7. increases fuel by correct amount; Marks for changes to other parts of program: 8. Returns `True` from `CheckCommandIsValid` if `burn` command was used; **Max 7 marks** if code contains errors	**8**
14	2	**Mark is for AO3 (evaluate)** ****** SCREEN CAPTURE ****** *Must match code from **14.1**.* *Code for **14.1** must be sensible.* Screen capture(s) showing that Player One's lumber has decreased by the same amount as their fuel has increased; **Notes for examiners:** due to random numbers in game exact values can vary; screen capture could show R or S below the B in the top-left corner of the grid; lumber and fuel both had an initial value of 10.  	**1**

Question			Marks
15	1	**All marks for AO3 (programming)** 1. Created new method called `GetFogOfWar`; **R.** other names for method **I.** case and minor typos 2. Method returns a Boolean value and takes the index of a tile as a parameter; **A.** alternatives to passing index of tile eg tile itself **I.** other parameters 3. Check to see if tile passed as parameter to method contains a piece belonging to the active player; 4. Gets all the neighbours of the tile passed as a parameter to the method; 5. Gets all the neighbours of the tiles identified in mark point 4 // gets all the neighbours of the tiles indentified in mark point 4 not already got; 6. Checks at least one neigbouring tile contains a piece belonging to the active player; 7. Iterative structure that looks at each tile identified as being within two of the tile passed to the method; **A.** not all tiles identified correctly 8. Every time a tile is checked the `PieceID` in the tile is obtained; 9. Returns a value of `False` if it correctly identifies, for the tiles checked, that the tile contains a piece belonging to the active player; 10. Method `GetFogOfWar` returns the correct value under all circumstances; 11. Modified `GetPieceTypeInTile` so that it calls `GetFogOfWar`; **A.** alternative identifier used as long as match that used for mark point 1 12. `GetPieceTypeInTile` returns a space character if the value returned by `GetFogOfWar` is `True`; 13. `GetPieceTypeInTile` returns the piece in the tile if there is a piece in the tile and a space character if either there is not a piece in the tile or when the value returned by `GetFogOfWar` is `True`; **R.** if no attempt for either mark points 11 or 12 **Alternative answer for mark points 4, 5 and 7** 4. Iterative structure that is used to check every tile; 5. Gets the distance of each tile from the tile passed as a parameter to the method; 7. Gets all tiles within a distance of two from the tile passed as a parameter to the method; **Note:** award mark points 4, 5 and 7 (both methods) for solutions where loop could terminate early if value of false is returned due to identification of a tile containing the player's piece that is within distance of two from tile passed as a parameter to the method, before all tiles that need to be checked have been identified. **Max 12** if code contains errors or if other parts of the subroutine `GetPieceTypeInTile` no longer work correctly	13
15	2	**Mark is for AO3 (evaluate)** ****** SCREEN CAPTURE ****** *Must match code from **15.1**, including prompts on screen capture matching those in code.* *Code for **15.1** must be sensible.*	1

		Screen capture(s) showing that for the game from game1.txt the grid is displayed correctly at the start of both player's turns; 1. Default game 2. Load game Q. Quit Enter your choice: 2 Enter the name of the file to load: game1.txt Player One current state - VPs: 0 Pieces in supply: 2 Lumber: 5 Fuel: 5 Player Two current state - VPs: 0 Pieces in supply: 2 Lumber: 5 Fuel: 5 __ __ __ /#B__/#b__/~ __ __/~L__/ __/ \ / __/~ __/ __/ __/ __/ __/ \ / __/#l__/# __/ __/#L__/~ __/~S\ __/ __/ __/ Player One state your three commands, pressing enter after each one. Enter command: Enter command: Enter command: Invalid command Invalid command Invalid command Player One current state - VPs: 1 Pieces in supply: 2 Lumber: 5 Fuel: 5 Player Two current state - VPs: 5 Pieces in supply: 2 Lumber: 5 Fuel: 5 Press Enter to continue... __ __ __ /#B__/#b__/~l__ __/~ __/ __/ \ / __/~ __/ __/ __/ __/ __/ \ / __/#l__/# __/ __/#L__/~ __/~ \ __/ __/ __/ Player Two state your three commands, pressing enter after each one. Enter command:	

VB.Net

Question			Marks
07	1		12

```
Console.Write("Enter value for n: ")
Dim Number As Integer = Console.ReadLine
Dim NumbersFoundSoFar As Integer = 0
Dim CurrentNumber As Integer = 1
Dim SumOfDigits As Integer
While NumbersFoundSoFar <> Number
  SumOfDigits = 0
  Dim NumAsString As String = CStr(CurrentNumber)
  For count = 0 To NumAsString.Length - 1
    SumOfDigits += Val(NumAsString(count))
  Next
  If CurrentNumber Mod SumOfDigits = 0 Then
    NumbersFoundSoFar += 1
    If NumbersFoundSoFar = Number Then
      Console.WriteLine(CurrentNumber)
    End If
  End If
  CurrentNumber += 1
End While
```

Alternative answer

```
Console.Write("Enter value for n: ")
Dim Number As Integer = Console.ReadLine
Dim NumbersFoundSoFar As Integer = 0
Dim CurrentNumber As Integer = 1
Dim SumOfDigits As Integer
While NumbersFoundSoFar <> Number
  Dim Temp As Integer = CurrentNumber
  SumOfDigits = 0
  While Temp > 0
    SumOfDigits += Temp Mod 10
    Temp \= 10
  End While
  If CurrentNumber Mod SumOfDigits = 0 Then
    NumbersFoundSoFar += 1
    If NumbersFoundSoFar = Number Then
      Console.WriteLine(CurrentNumber)
    End If
  End If
  CurrentNumber += 1
End While
```

Recursive answer

```
Function SumDigits(ByVal Num As Integer) As Integer
  Dim Sum As Integer
  If Num = 0 Then
    Sum = 0
  Else
    Sum = Num Mod 10 + SumDigits(Num\10)
  End If
  Return Sum
```

```
End Function


Sub Main()
  Console.Write("Enter value for n: ")
  Dim n As Integer = Console.ReadLine()
  Dim NthHarshad As Integer = 0
  Dim Counter As Integer = 1
  Dim Value As Integer
  While n <> NthHarshad
    Value = SumDigits(Counter)
    If Counter Mod Value = 0 Then
      NthHarshad += 1
    End If
    If n = NthHarshad Then
      Console.WriteLine(Counter)
    End If
    Counter = Counter + 1
  End While
End Sub
```

12 **1** **5**

```
Public Function DestroyPiecesAndCountVPs(ByRef Player1VPs As
Integer, ByRef Player2VPs As Integer) As Boolean
  Dim BaronDestroyed As Boolean = False
  Dim ListOfTilesContainingDestroyedPieces As New List(Of Tile)
  For Each T In Tiles
    If T.GetPieceInTile() IsNotNothing Then
      Dim ListOfNeighbours As List(Of Tile) = New List(Of
Tile)(T.GetNeighbours())
      Dim NoOfConnections As Integer = 0
      For Each N In ListOfNeighbours
        If N.GetPieceInTile() IsNot Nothing Then
          NoOfConnections += 1
        End If
      Next
      Dim ThePiece As Piece = T.GetPieceInTile()
      If NoOfConnections >= ThePiece.GetConnectionsNeededToDestroy()
Then
        ThePiece.DestroyPiece()
        If ThePiece.GetPieceType().ToUpper() = "B" Then
          BaronDestroyed = True
        End If
        ListOfTilesContainingDestroyedPieces.Add(T)
        If ThePiece.GetBelongsToPlayer1() Then
          Player2VPs += ThePiece.GetVPs()
        Else
          Player1VPs += ThePiece.GetVPs()
        End If
      Else
        If ThePiece.GetPieceType() = "L" Then
          Player1VPs += 1
        ElseIf ThePiece.GetPieceType() = "l" Then
          Player2VPs += 1
        End If
      End If
    End If
  Next
  For Each T In ListOfTilesContainingDestroyedPieces
    T.SetPiece(Nothing)
  Next
```

```
  Return BaronDestroyed
End Function
```

Alternative answer

```
      Else
       If ThePiece.GetBelongsToPlayer1() And ThePiece.GetPieceType()
= "L" Then
         Player1VPs += 1
       ElseIf Not ThePiece.GetBelongsToPlayer1() And
ThePiece.GetPieceType() = "l" Then
         Player2VPs += 1
```

13 **1** **7**

```
Class RangerPiece
  Inherits Piece
  Public Sub New(ByVal Player1 As Boolean)
    MyBase.New(Player1)
    PieceType = "R"
  End Sub

  Public Overrides Function CheckMoveIsValid(ByVal
DistanceBetweenTiles As Integer, ByVal StartTerrain As String, ByVal
EndTerrain As String) As Integer
    If DistanceBetweenTiles = 1 Then
      If StartTerrain = "~" Or EndTerrain = "~" Then
        Return FuelCostOfMove * 2
      Else
        Return FuelCostOfMove
      End If
    End If
    If StartTerrain = "#" And EndTerrain = "#" Then
      Return FuelCostOfMove
    End If
    Return -1
  End Function
End Class
```

```
Public Sub AddPiece(ByVal BelongsToPlayer1 As Boolean, ByVal
TypeOfPiece As String, ByVal Location As Integer)
  Dim NewPiece As Piece
  If TypeOfPiece = "Baron" Then
    NewPiece = New BaronPiece(BelongsToPlayer1)
  ElseIf TypeOfPiece = "LESS" Then
    NewPiece = New LESSPiece(BelongsToPlayer1)
  ElseIf TypeOfPiece = "PBDS" Then
    NewPiece = New PBDSPiece(BelongsToPlayer1)
  ElseIf TypeOfPiece = "Ranger" Then
    NewPiece = New RangerPiece(BelongsToPlayer1)
  Else
    NewPiece = New Piece(BelongsToPlayer1)
  End If
  Pieces.Add(NewPiece)
  Tiles(Location).SetPiece(NewPiece)
End Sub
```

Alternative answer

```
  Public Overrides Function CheckMoveIsValid(ByVal
DistanceBetweenTiles As Integer, ByVal StartTerrain As String, ByVal
EndTerrain As String) As Integer
    If StartTerrain = "#" And EndTerrain = "#" Then
      Return FuelCostOfMove
    End If
    Return MyBase.CheckMoveIsValid(DistanceBetweenTiles,
StartTerrain, EndTerrain)
  End Function
```

14 **1** **8**

```
Function CheckCommandIsValid(ByVal Items As List(Of String)) As
Boolean
  If Items.Count > 0 Then
    Select Case Items(0)
      Case "move"
        Return CheckMoveCommandFormat(Items)
      Case "dig", "saw", "spawn"
        Return CheckStandardCommandFormat(Items)
      Case "move"
        Return CheckUpgradeCommandFormat(Items)
      Case "burn"
        Return True
    End Select
  End If
  Return False
End Function

Public Function ExecuteCommand(ByVal Items As List(Of String), ByRef
FuelChange As Integer, ByRef LumberChange As Integer, ByRef
SupplyChange As Integer, ByVal FuelAvailable As Integer, ByVal
LumberAvailable As Integer, ByVal TilesInSupply As Integer) As
String
  Select Case Items(0)
    Case "move"
    ..
    Case "burn"
      If LumberAvailable < 1 Then
        Return "Cannot burn lumber"
      End If
      Dim Rno As Integer = RNoGen.NextDouble() * (LumberAvailable -
1) + 1
      LumberChange = -Rno
      FuelChange = Rno
    End Select
  Return "Command executed"
End Function
```

Alternative answer

```
Dim Rno As Integer = Rnd() * (LumberAvailable - 1) + 1
```

15 **1**

```
Private Function GetFogOfWar(ByVal ID As Integer) As Boolean
  Dim ListOfNeighbours As List(Of Tile) = New List(Of
Tile)(Tiles(ID).GetNeighbours())
```

13

```
  Dim ListToCheck As List(Of Tile) = New List(Of
Tile)(Tiles(ID).GetNeighbours())
  ListToCheck.Add(Tiles(ID))
  For Each N In ListOfNeighbours
    ListToCheck.AddRange(N.GetNeighbours())
  Next
  For Each N In ListToCheck
    Dim ThePiece As Piece = N.GetPieceInTile()
    If ThePiece IsNot Nothing Then
      If ThePiece.GetBelongsToPlayer1() = Player1Turn Then
        Return False
      End If
    End If
  Next
  Return True
End Function

Public Function GetPieceTypeInTile(ByVal ID As Integer) As String
  Dim ThePiece As Piece = Tiles(ID).GetPieceInTile()
  If PieceInTile Is Nothing Then
    Return " "
  Else
    If GetFogOfWar(ID) Then
      Return " "
    End If
    Return ThePiece.GetPieceType()
  End If
End Function
```

Python2

Question			Marks

07	1	`Number = int(raw_input("Enter value for n: "))`
		`NumbersFoundSoFar = 0`
		`CurrentNumber = 1`
		`while NumbersFoundSoFar != Number:`
		`  SumOfDigits = 0`
		`  NumAsString = str(CurrentNumber)`
		`  for count in range (0, len(NumAsString)):`
		`  SumOfDigits += int(NumAsString[count])`
		`  if CurrentNumber % SumOfDigits == 0:`
		`  NumbersFoundSoFar += 1`
		`  if NumbersFoundSoFar == Number:`
		`  print CurrentNumber`
		`  CurrentNumber += 1`
		
		Alternative answer
		
		`Number = int(raw_input("Enter value for n: "))`
		`NumbersFoundSoFar = 0`
		`CurrentNumber = 1`
		`while NumbersFoundSoFar != Number:`
		`  Temp = CurrentNumber`
		`  SumOfDigits = 0`
		`  while Temp > 0:`
		`  SumOfDigits += Temp % 10`
		`  Temp = Temp // 10`
		`  if CurrentNumber % SumOfDigits == 0:`
		`  NumbersFoundSoFar += 1`
		`  if NumbersFoundSoFar == Number:`
		`  print CurrentNumber`
		`  CurrentNumber += 1`
		
		Recursive answer
		
		`def SumDigits(Num):`
		`  if Num == 0:`
		`  Sum = 0`
		`  else:`
		`  Sum = Num % 10 + SumDigits(Num//10)`
		`  return Sum`
		
		`n = int(raw_input("Enter value for n: "))`
		`NthHarshad = 0`
		`Counter = 1`
		`while n != NthHarshad:`
		`  Value = SumDigits(Counter)`
		`  if Counter % Value == 0:`
		`  NthHarshad += 1`
		`  if n == NthHarshad:`
		`  print Counter`
		`  Counter = Counter + 1`
			12

12	1	`def DestroyPiecesAndCountVPs(self):`	5

```
  BaronDestroyed = False
  Player1VPs = 0
  Player2VPs = 0
  ListOfTilesContainingDestroyedPieces = []
  for T in self._Tiles:
    if T.GetPieceInTile() is not None:
      ListOfNeighbours = T.GetNeighbours()
      NoOfConnections = 0
      for N in ListOfNeighbours:
        if N.GetPieceInTile() is not None:
          NoOfConnections += 1
      ThePiece = T.GetPieceInTile()
      if NoOfConnections >=
ThePiece.GetConnectionsNeededToDestroy():
        ThePiece.DestroyPiece()
        if ThePiece.GetPieceType().upper() == "B":
          BaronDestroyed = True
        ListOfTilesContainingDestroyedPieces.append(T)
        if ThePiece.GetBelongsToPlayer1():
          Player2VPs += ThePiece.GetVPs()
        else:
          Player1VPs += ThePiece.GetVPs()
      else:
        if ThePiece.GetPieceType() == "L":
          Player1VPs += 1
        elif ThePiece.GetPieceType() == "l":
          Player2VPs += 1
  for T in ListOfTilesContainingDestroyedPieces:
    T.SetPiece(None)
  return BaronDestroyed, Player1VPs, Player2VPs
```

Alternative answer

```
      else:
       if ThePiece.GetBelongsToPlayer1() and ThePiece.GetPieceType()
== "L":
         Player1VPs += 1
       elif not ThePiece.GetBelongsToPlayer1() and
ThePiece.GetPieceType() == "l":
         Player2VPs += 1
```

13	**1**	**7**

```
class RangerPiece(Piece):
  def __init__(self, Player1):
    super(RangerPiece, self).__init__(Player1)
    self._PieceType = "R"

  def CheckMoveIsValid(self, DistanceBetweenTiles, StartTerrain,
EndTerrain):
    if DistanceBetweenTiles == 1:
      if StartTerrain == "~" or EndTerrain == "~":
        return self._FuelCostOfMove * 2
      else:
        return self._FuelCostOfMove
    if StartTerrain == "#" and EndTerrain == "#":
      return self._FuelCostOfMove
    return -1

def AddPiece(self, BelongsToPlayer1, TypeOfPiece, Location):
```

```
  if TypeOfPiece == "Baron":
    NewPiece = BaronPiece(BelongsToPlayer1)
  elif TypeOfPiece == "LESS":
    NewPiece = LESSPiece(BelongsToPlayer1)
  elif TypeOfPiece == "PBDS":
    NewPiece = PBDSPiece(BelongsToPlayer1)
  elif TypeOfPiece == "Ranger":
    NewPiece = RangerPiece(BelongsToPlayer1)
  else:
    NewPiece = Piece(BelongsToPlayer1)
  self._Pieces.append(NewPiece)
  self._Tiles[Location].SetPiece(NewPiece)
```

Alternative answer

```
def CheckMoveIsValid(self, DistanceBetweenTiles, StartTerrain,
EndTerrain):
  if StartTerrain == "#" and EndTerrain == "#":
    return self._FuelCostOfMove
  return super(RangerPiece,
self).CheckMoveIsValid(DistanceBetweenTiles, StartTerrain,
EndTerrain)
```

14 **1**

```
def CheckCommandIsValid(Items):
  if len(Items) > 0:
    if Items[0] == "move":
      return CheckMoveCommandformat(Items)
    elif Items[0] in ["dig", "saw", "spawn"]:
      return CheckStandardCommandformat(Items)
    elif Items[0] == "upgrade":
      return CheckUpgradeCommandformat(Items)
    elif Items[0] == "burn":
      return True
  return False

def ExecuteCommand(self, Items, FuelAvailable, LumberAvailable,
PiecesInSupply):
  FuelChange = 0
  LumberChange = 0
  SupplyChange = 0
  if Items[0] == "move":
...
  elif Items[0] == "burn":
    if LumberAvailable < 1:
      return "Cannot burn lumber"
    Rno = random.randint(1, LumberAvailable)
    LumberChange = -Rno
    FuelChange = Rno
  return "Command executed", FuelChange, LumberChange, SupplyChange
```

8

15 **1**

```
def GetFogOfWar(self, ID):
  ListOfNeighbours = []
  ListOfNeighbours.extend(self._Tiles[ID].GetNeighbours())
  ListToCheck = []
  ListToCheck.extend(self._Tiles[ID].GetNeighbours())
  ListToCheck.append(self._Tiles[ID])
  for N in ListOfNeighbours:
    ListToCheck.extend(N.GetNeighbours())
```

13

```
  for N in ListToCheck:
    ThePiece = N.GetPieceInTile()
    if ThePiece is not None:
      if ThePiece.GetBelongsToPlayer1() == self._Player1Turn:
        return False
  return True

def GetPieceTypeInTile(self, ID):
  ThePiece = self._Tiles[ID].GetPieceInTile()
  if ThePiece is None:
    return " "
  else:
    if self.GetFogOfWar(ID):
      return " "
    return ThePiece.GetPieceType()
```

Python3

<table>
<tr><th colspan="2">Question</th><th></th><th>Marks</th></tr>
<tr><td>07</td><td>1</td><td>

```
Number = int(input("Enter value for n: "))
NumbersFoundSoFar = 0
CurrentNumber = 1
while NumbersFoundSoFar != Number:
  SumOfDigits = 0
  NumAsString = str(CurrentNumber)
  for count in range (0, len(NumAsString)):
    SumOfDigits += int(NumAsString[count])
  if CurrentNumber % SumOfDigits == 0:
    NumbersFoundSoFar += 1
    if NumbersFoundSoFar == Number:
      print(CurrentNumber)
  CurrentNumber += 1
```

Alternative answer

```
Number = int(input("Enter value for n: "))
NumbersFoundSoFar = 0
CurrentNumber = 1
while NumbersFoundSoFar != Number:
  Temp = CurrentNumber
  SumOfDigits = 0
  while Temp > 0:
    SumOfDigits += Temp % 10
    Temp = Temp // 10
  if CurrentNumber % SumOfDigits == 0:
    NumbersFoundSoFar += 1
    if NumbersFoundSoFar == Number:
      print(CurrentNumber)
  CurrentNumber += 1
```

Recursive answer

```
def SumDigits(Num):
  if Num == 0:
    Sum = 0
  else:
    Sum = Num % 10 + SumDigits(Num//10)
  return Sum

n = int(input("Enter value for n: "))
NthHarshad = 0
Counter = 1
while n != NthHarshad:
  Value = SumDigits(Counter)
  if Counter % Value == 0:
    NthHarshad += 1
  if n == NthHarshad:
    print(Counter)
  Counter = Counter + 1
```

</td><td>12</td></tr>
<tr><td>12</td><td>1</td><td>

```
def DestroyPiecesAndCountVPs(self):
```

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

```
  BaronDestroyed = False
  Player1VPs = 0
  Player2VPs = 0
  ListOfTilesContainingDestroyedPieces = []
  for T in self._Tiles:
    if T.GetPieceInTile() is not None:
      ListOfNeighbours = T.GetNeighbours()
      NoOfConnections = 0
      for N in ListOfNeighbours:
        if N.GetPieceInTile() is not None:
          NoOfConnections += 1
      ThePiece = T.GetPieceInTile()
      if NoOfConnections >=
ThePiece.GetConnectionsNeededToDestroy():
        ThePiece.DestroyPiece()
        if ThePiece.GetPieceType().upper() == "B":
          BaronDestroyed = True
        ListOfTilesContainingDestroyedPieces.append(T)
        if ThePiece.GetBelongsToPlayer1():
          Player2VPs += ThePiece.GetVPs()
        else:
          Player1VPs += ThePiece.GetVPs()
      else:
        if ThePiece.GetPieceType() == "L":
          Player1VPs += 1
        elif ThePiece.GetPieceType() == "l":
          Player2VPs += 1
  for T in ListOfTilesContainingDestroyedPieces:
    T.SetPiece(None)
  return BaronDestroyed, Player1VPs, Player2VPs
```

Alternative answer

```
      else:
       if ThePiece.GetBelongsToPlayer1() and ThePiece.GetPieceType()
== "L":
         Player1VPs += 1
       elif not ThePiece.GetBelongsToPlayer1() and
ThePiece.GetPieceType() == "l":
         Player2VPs += 1
```

13	**1**	**7**

```
class RangerPiece(Piece):
  def __init__(self, Player1):
    super(RangerPiece, self).__init__(Player1)
    self._PieceType = "R"

  def CheckMoveIsValid(self, DistanceBetweenTiles, StartTerrain,
EndTerrain):
    if DistanceBetweenTiles == 1:
      if StartTerrain == "~" or EndTerrain == "~":
        return self._FuelCostOfMove * 2
      else:
        return self._FuelCostOfMove
    if StartTerrain == "#" and EndTerrain == "#":
      return self._FuelCostOfMove
    return -1

def AddPiece(self, BelongsToPlayer1, TypeOfPiece, Location):
```

```
  if TypeOfPiece == "Baron":
    NewPiece = BaronPiece(BelongsToPlayer1)
  elif TypeOfPiece == "LESS":
    NewPiece = LESSPiece(BelongsToPlayer1)
  elif TypeOfPiece == "PBDS":
    NewPiece = PBDSPiece(BelongsToPlayer1)
  elif TypeOfPiece == "Ranger":
    NewPiece = RangerPiece(BelongsToPlayer1)
  else:
    NewPiece = Piece(BelongsToPlayer1)
  self._Pieces.append(NewPiece)
  self._Tiles[Location].SetPiece(NewPiece)
```

Alternative answer

```
def CheckMoveIsValid(self, DistanceBetweenTiles, StartTerrain,
EndTerrain):
  if StartTerrain == "#" and EndTerrain == "#":
    return self._FuelCostOfMove
  return super(RangerPiece,
self).CheckMoveIsValid(DistanceBetweenTiles, StartTerrain,
EndTerrain)
```

14 **1** **8**

```
def CheckCommandIsValid(Items):
  if len(Items) > 0:
    if Items[0] == "move":
      return CheckMoveCommandformat(Items)
    elif Items[0] in ["dig", "saw", "spawn"]:
      return CheckStandardCommandformat(Items)
    elif Items[0] == "upgrade":
      return CheckUpgradeCommandformat(Items)
    elif Items[0] == "burn":
      return True
  return False

def ExecuteCommand(self, Items, FuelAvailable, LumberAvailable,
PiecesInSupply):
  FuelChange = 0
  LumberChange = 0
  SupplyChange = 0
  if Items[0] == "move":
...
  elif Items[0] == "burn":
    if LumberAvailable < 1:
      return "Cannot burn lumber"
    Rno = random.randint(1, LumberAvailable)
    LumberChange = -Rno
    FuelChange = Rno
  return "Command executed", FuelChange, LumberChange, SupplyChange
```

15 **1** **13**

```
def GetFogOfWar(self, ID):
  ListOfNeighbours = []
  ListOfNeighbours.extend(self._Tiles[ID].GetNeighbours())
  ListToCheck = []
  ListToCheck.extend(self._Tiles[ID].GetNeighbours())
  ListToCheck.append(self._Tiles[ID])
  for N in ListOfNeighbours:
    ListToCheck.extend(N.GetNeighbours())
```

```
  for N in ListToCheck:
    ThePiece = N.GetPieceInTile()
    if ThePiece is not None:
      if ThePiece.GetBelongsToPlayer1() == self._Player1Turn:
        return False
  return True

def GetPieceTypeInTile(self, ID):
  ThePiece = self._Tiles[ID].GetPieceInTile()
  if ThePiece is None:
    return " "
  else:
    if self.GetFogOfWar(ID):
      return " "
    return ThePiece.GetPieceType()
```

C#

Question			Marks
07	1		12

```
Console.Write("Enter value for n: ");
int number = Convert.ToInt32(Console.ReadLine());
int numbersFoundSoFar = 0;
int currentNumber = 1;
int sumOfDigits;
while ((numbersFoundSoFar != number))
{
    int temp = currentNumber;
    sumOfDigits = 0;
    string numAsString = currentNumber.ToString();
    for (int count = 0; (count <= (numAsString.Length - 1));
count++)
    {
        sumOfDigits = (sumOfDigits +
Convert.ToInt32(numAsString[count].ToString()));
    }

    if (((currentNumber % sumOfDigits) == 0))
    {
        numbersFoundSoFar++;
        if ((numbersFoundSoFar == number))
        {
            Console.WriteLine(currentNumber);
        }

    }
    currentNumber++;
}
Console.ReadLine();
```

Alternative answer

```
Console.Write("Enter value for n: ");
int number = Convert.ToInt32(Console.ReadLine());
int numbersFoundSoFar = 0;
int currentNumber = 1;
int sumOfDigits;
while ((numbersFoundSoFar != number))
{
    int temp = currentNumber;
    sumOfDigits = 0;
    while (temp > 0)
    {
        sumOfDigits +=temp % 10;
        temp = temp / 10;
    }
    if (currentNumber % sumOfDigits == 0)
    {
        numbersFoundSoFar++;
        if (numbersFoundSoFar == number)
        {
            Console.WriteLine(currentNumber);
        }

    }
```

```
    currentNumber++;
}
Console.ReadLine();
```

Recursive answer

```
static int SumDigits(int num)
{
    int sum;
    if (num == 0)
    {
        sum = 0;
    }
    else
    {
        sum = num % 10 + SumDigits(num / 10);
    }
    return sum;
}

static void Main(string[] args)
{
   Console.Write("Enter value for n: ");
    int n = Convert.ToInt32(Console.ReadLine());
    int nthHarshad = 0;
    int counter = 1;
    int value;
    while ((n != nthHarshad))
    {
        value = SumDigits(counter);
        if ((counter % value) == 0)
        {
            nthHarshad++;
        }
        if ((n == nthHarshad))
        {
            Console.WriteLine(counter);
        }
        counter = (counter + 1);
    }
    Console.ReadLine();
}
```

12	**1**	`public bool DestroyPiecesAndCountVPs(ref int player1VPs, ref int player2VPs)`
		`{`
		`    bool baronDestroyed = false;`
		`    List<Tile> listOfTilesContainingDestroyedPieces = new List<Tile>();`
		`    foreach (var t in tiles)`
		`    {`
		`        if (t.GetPieceInTile() != null)`
		`        {`
		`            List<Tile> listOfNeighbours = new List<Tile>(t.GetNeighbours());`
		`            int noOfConnections = 0;`
		`            foreach (var n in listOfNeighbours)`
			5

```
            {
                if (n.GetPieceInTile() != null)
                {
                    noOfConnections += 1;
                }
            }
            Piece thePiece = t.GetPieceInTile();
            if (noOfConnections >=
thePiece.GetConnectionsNeededToDestroy())
            {
                thePiece.DestroyPiece();
                if (thePiece.GetPieceType().ToUpper() == "B")
                    baronDestroyed = true;
                listOfTilesContainingDestroyedPieces.Add(t);
                if (thePiece.GetBelongsToPlayer1())
                {
                    player2VPs += thePiece.GetVPs();
                }
                else
                {
                    player1VPs += thePiece.GetVPs();
                }
            }
            else
            {
                if (thePiece.GetPieceType() == "L")
                {
                    player1VPs += 1;
                }
                else if (thePiece.GetPieceType() == "l")
                {
                    player2VPs += 1;
                }
            }
        }
    }
    foreach (var t in listOfTilesContainingDestroyedPieces)
    {
        t.SetPiece(null);
    }
    return baronDestroyed;
}
```

Alternative answer

```
else
{
    if (thePiece.GetBelongsToPlayer1() && thePiece.GetPieceType() ==
"L" )
    {
        player1VPs += 1;
    }
    else if (!thePiece.GetBelongsToPlayer1() &&
thePiece.GetPieceType() == "l")
    {
        player2VPs += 1;
    }
```

13	1	**class RangerPiece : Piece**
		{
		public RangerPiece(bool player1)
		:base(player1)
		{
		pieceType = "R";
		}
		
		public override int CheckMoveIsValid(int distanceBetweenTiles, string startTerrain, string endTerrain)
		{
		if (distanceBetweenTiles == 1)
		{
		if (startTerrain == "~" || endTerrain == "~")
		{
		return fuelCostOfMove * 2;
		}
		else
		{
		return fuelCostOfMove;
		}
		}
		if (startTerrain == "#" && endTerrain == "#")
		{
		return fuelCostOfMove;
		}
		return -1;
		}
		}
		
		public void AddPiece(bool belongsToPlayer1, string typeOfPiece, int location)
		{
		Piece newPiece;
		if (typeOfPiece == "Baron")
		{
		newPiece = new BaronPiece(belongsToPlayer1);
		}
		else if (typeOfPiece == "LESS")
		{
		newPiece = new LESSPiece(belongsToPlayer1);
		}
		else if (typeOfPiece == "PBDS")
		{
		newPiece = new PBDSPiece(belongsToPlayer1);
		}
		else if (typeOfPiece == "Ranger")
		{
		newPiece = new RangerPiece(belongsToPlayer1);
		}
		else
		{
		newPiece = new Piece(belongsToPlayer1);
		}
		pieces.Add(newPiece);
		tiles[location].SetPiece(newPiece);
		}
			7

Alternative answer

```
public override int CheckMoveIsValid(int distanceBetweenTiles,
string startTerrain, string endTerrain)
{
    if (startTerrain == "#" && endTerrain == "#")
    {
        return fuelCostOfMove;
    }
    return
base.CheckMoveIsValid(distanceBetweenTiles,startTerrain,endTerrain);
}
```

14 **1** **8**

```
public static bool CheckCommandIsValid(List<string> items)
{
    if (items.Count > 0)
    {
        switch (items[0])
        {
            case "move":
                {
                    return CheckMoveCommandFormat(items);
                }

            case "dig":
            case "saw":
            case "spawn":
                {
                    return CheckStandardCommandFormat(items);
                }

            case "upgrade":
                {
                    return CheckUpgradeCommandFormat(items);
                }
            case "burn":
                {
                    return true;
                }
        }
    }
    return false;
}

public string ExecuteCommand(List<string> items, ref int fuelChange,
ref int lumberChange, ref int supplyChange, int fuelAvailable, int
lumberAvailable, int piecesInSupply)
{
    switch (items[0])
    {
        case "move":
        ...
        case "burn":
            {
                Random rnd = new Random();
                if (lumberAvailable < 1)
                {
                    return "Cannot burn lumber";
                }
```

```
                int rno = rnd.Next(1, lumberAvailable + 1);
                lumberChange = -rno;
                fuelChange = rno;
                break;
            }
    }
    return "Command executed";
}
```

15 **1**

```
private bool GetFogOfWar(int ID)
{
    List<Tile> listOfNeighbours = new
List<Tile>(tiles[ID].GetNeighbours());
    List<Tile> ListToCheck = new
List<Tile>(tiles[ID].GetNeighbours());
    ListToCheck.Add(tiles[ID]);
    foreach (var n in listOfNeighbours)
    {
        ListToCheck.AddRange(n.GetNeighbours());
    }
    foreach (var n in ListToCheck)
    {
        Piece thePiece = n.GetPieceInTile();
        if (thePiece != null)
        {
            if (thePiece.GetBelongsToPlayer1() == player1Turn)
            {
                return false;
            }
        }
    }
    return true;
}

public string GetPieceTypeInTile(int ID)
{
    Piece thePiece = tiles[ID].GetPieceInTile();
    if (thePiece == null)
    {
        return " ";
    }
    else
    {
        if (GetFogOfWar(ID))
        {
            return " ";
        }
        return thePiece.GetPieceType();
    }
}
```

13

PASCAL/Delphi

Question			Marks
07	1		12

```
var
  Number : integer;
  NumbersFoundSoFar : integer;
  CurrentNumber : integer;
  SumOfDigits : integer;
  NumAsString : string;
  Count : integer;
begin
  write('Enter value for n: ');
  readln(Number);
  NumbersFoundSoFar := 0;
  CurrentNumber := 1;
  while (NumbersFoundSoFar <> Number) do
  begin
    SumOfDigits := 0;
    NumAsString := inttostr(CurrentNumber);
    for Count := 1 to length(NumAsString) do
      SumOfDigits := SumOfDigits + strtoint(NumAsString[Count]);
    if CurrentNumber mod SumOfDigits = 0 then
    begin
      inc(NumbersFoundSoFar);
      if NumbersFoundSoFar = Number then
         writeln(CurrentNumber);
    end;
    inc(CurrentNumber);
  end;
  readln;
end.
```

Alternative answer

```
var
  Number : integer;
  NumbersFoundSoFar : integer;
  CurrentNumber : integer;
  SumOfDigits : integer;
  Temp : integer;
begin
  write('Enter value for n: ');
  readln(Number);
  NumbersFoundSoFar := 0;
  CurrentNumber := 1;
  while NumbersFoundSoFar <> Number do
  begin
    Temp := CurrentNumber;
    SumOfDigits := 0;
    while Temp > 0 do
    begin
```

```
      SumOfDigits := SumOfDigits + (Temp mod 10);
      Temp := Temp div 10;
    end;
    if CurrentNumber mod SumOfDigits = 0 then
    begin
      inc(NumbersFoundSoFar);
      if NumbersFoundSoFar = Number then
        writeln(inttostr(CurrentNumber));
    end;
    inc(CurrentNumber);
  end;
  readln;
end.
```

Recursive answer

```
function SumDigits(Num : integer) : integer;
var
  Sum : integer;
begin
  if Num = 0 then
    Sum := 0
  else
    Sum := (Num mod 10) + SumDigits(Num div 10);
  SumDigits := Sum;
end;

var
  N, NthHarshad, Counter, Value : integer;
begin
  write('Enter value for n: ');
  readln(N);
  NthHarshad := 0;
  Counter := 1;
  while (N <> NthHarshad) do
  begin
    Value := SumDigits(Counter);
    if Counter mod Value = 0 then
      inc(NthHarshad);
    if N = NthHarshad then
      writeln(inttostr(Counter));
    inc(Counter);
  end;
  readln;
end.
```

12	**1**	`function HexGrid.DestroyPiecesAndCountVPs(var Player1VPs: integer;`
		`  var Player2VPs: integer): boolean;`
		`var`
		`  BaronDestroyed: boolean;`
		`  ListOfTilesContainingDestroyedPieces: TTileArray;`
			5

```
  ListOfNeighbours: TTileArray;
  NoOfConnections: integer;
  ThePiece: Piece;
  N: integer;
  T: integer;
begin
  BaronDestroyed := false;
  setlength(ListOfTilesContainingDestroyedPieces, 0);
  for T := low(Tiles) to high(Tiles) do
  begin
    if not(Tiles[T].GetPieceInTile() = nil) then
    begin
      ListOfNeighbours := Tiles[T].GetNeighbours();
      NoOfConnections := 0;
      for N := low(ListOfNeighbours) to high(ListOfNeighbours) do
        if not(ListOfNeighbours[N].GetPieceInTile() = nil) then
          inc(NoOfConnections);
      ThePiece := Tiles[T].GetPieceInTile();
      if (NoOfConnections >=
ThePiece.GetConnectionsNeededToDestroy()) then
      begin
        ThePiece.DestroyPiece();
        if uppercase(ThePiece.GetPieceType()) = 'B' then
          BaronDestroyed := true;
        setlength(ListOfTilesContainingDestroyedPieces,
          length(ListOfTilesContainingDestroyedPieces) + 1);
        ListOfTilesContainingDestroyedPieces
          [high(ListOfTilesContainingDestroyedPieces)] := Tiles[T];
        if ThePiece.GetBelongsToPlayer1() then
          Player2VPs := Player2VPs + ThePiece.GetVPs()
        else
          Player1VPs := Player1VPs + ThePiece.GetVPs();
      end
      else
      begin
        if ThePiece.GetPieceType() = 'L' then
          inc(Player1VPs)
        else if ThePiece.GetPieceType() = 'l' then
          inc(Player2VPs);
      end;
    end;
  end;
  for T := low(ListOfTilesContainingDestroyedPieces)
    to high(ListOfTilesContainingDestroyedPieces) do
    ListOfTilesContainingDestroyedPieces[T].SetPiece(nil);
  DestroyPiecesAndCountVPs := BaronDestroyed;
end;
```

13 **1**

```
type
  RangerPiece = class(Piece)
  public
    constructor Init(Player1: boolean);
```

7

```
    function CheckMoveIsValid(DistanceBetweenTiles: integer;
      StartTerrain: string; EndTerrain: string): integer; override;
  end;

constructor RangerPiece.Init(Player1: boolean);
begin
  inherited;
  PieceType := 'R';
end;

function RangerPiece.CheckMoveIsValid(DistanceBetweenTiles: integer;
      StartTerrain: string; EndTerrain: string): integer;
begin
  if DistanceBetweenTiles = 1 then
  begin
    if (StartTerrain = '~') or (EndTerrain = '~') then
    begin
      CheckMoveIsValid := FuelCostOfMove * 2;
      exit;
    end
    else
    begin
      CheckMoveIsValid := FuelCostOfMove;
      exit;
    end;
  end;
  if (StartTerrain = '#') and (EndTerrain = '#') then
  begin
    CheckMoveIsValid := FuelCostOfMove;
    exit;
  end;
  CheckMoveIsValid := -1;
end;

procedure HexGrid.AddPiece(BelongsToPlayer1: boolean; TypeOfPiece:
string;
  Location: integer);
var
  NewPiece: Piece;
begin
  if TypeOfPiece = 'Baron' then
  begin
    NewPiece := BaronPiece.Init(BelongsToPlayer1);
  end
  else if TypeOfPiece = 'LESS' then
    NewPiece := LESSPiece.Init(BelongsToPlayer1)
  else if TypeOfPiece = 'PBDS' then
    NewPiece := PBDSPiece.Init(BelongsToPlayer1)
  else if TypeOfPiece = 'Ranger' then
    NewPiece := RangerPiece.Init(BelongsToPlayer1)
  else
    NewPiece := Piece.Init(BelongsToPlayer1);
  setlength(Pieces, (length(Pieces) + 1));
```

		Pieces[High(Pieces)] := NewPiece; Tiles[Location].SetPiece(NewPiece); end;	

14	**1**	function CheckCommandIsValid(Items: TStringArray) : boolean;
		begin
		if Length(Items) > 0 then
		begin
		if Items[0] = 'move' then
		begin
		CheckCommandIsValid := CheckMoveCommandFormat(Items);
		exit;
		end
		else if (Items[0] = 'dig') or (Items[0] = 'saw') or (Items[0] = 'spawn')
		then
		begin
		CheckCommandIsValid := CheckStandardCommandFormat(Items);
		exit;
		end
		else if Items[0] = 'upgrade' then
		begin
		CheckCommandIsValid := CheckUpgradeCommandFormat(Items);
		exit;
		end
		else if Items[0] = 'burn' then
		begin
		CheckCommandIsValid := true;
		exit;
		end;
		end;
		CheckCommandIsValid := false;
		end;
		
		function HexGrid.ExecuteCommand(Items: TStringArray; var FuelChange: integer;
		var LumberChange: integer; var SupplyChange: integer;
		FuelAvailable: integer;
		LumberAvailable: integer; PiecesInSupply: integer): string;
		var
		FuelCost: integer;
		LumberCost: integer;
		RNo : integer;
		begin
		if Items[0] = 'move' then
		begin
		FuelCost := ExecuteMoveCommand(Items, FuelAvailable);
		if FuelCost < 0 then
		begin
		ExecuteCommand := 'That move can''t be done';
		exit;
		end;
			8

```
    FuelChange := -FuelCost;
  end
  else if (Items[0] = 'saw') or (Items[0] = 'dig') then
  begin
    if not(ExecuteCommandInTile(Items, FuelChange, LumberChange))
then
    begin
      ExecuteCommand := 'Couldn''t do that';
      exit;
    end;
  end
  else if Items[0] = 'spawn' then
  begin
    LumberCost := ExecuteSpawnCommand(Items, LumberAvailable,
PiecesInSupply);
    if LumberCost < 0 then
    begin
      ExecuteCommand := 'Spawning did not occur';
      exit;
    end;
    LumberChange := -LumberCost;
    SupplyChange := 1;
  end
  else if Items[0] = 'upgrade' then
  begin
    LumberCost := ExecuteUpgradeCommand(Items, LumberAvailable);
    if LumberCost < 0 then
    begin
      ExecuteCommand := 'Upgrade not possible';
      exit;
    end;
    LumberChange := -LumberCost;
  end
  else if Items[0] = 'burn' then
  begin
    if LumberAvailable < 1 then
    begin
      ExecuteCommand := 'Cannot burn lumber';
      exit;
    end;
    RNo := trunc(random() * (LumberAvailable -1)) + 1;
    LumberChange := -RNo;
    FuelChange := RNo;
  end;
  ExecuteCommand := 'Command Executed';
end;
```

15 **1**

```
type
  HexGrid = class
  protected
    Tiles: TTileArray;
    Pieces: TPieceArray;
    Size: integer;
```

13

```
    Player1Turn: boolean;
  public
    constructor Init(N: integer);
    procedure SetUpGridTerrain(ListOfTerrain: TStringArray);
    procedure AddPiece(BelongsToPlayer1: boolean; TypeOfPiece:
string;
      Location: integer);
    function ExecuteCommand(Items: TStringArray; var FuelChange:
integer;
      var LumberChange: integer; var SupplyChange: integer;
      FuelAvailable: integer; LumberAvailable: integer;
      PiecesInSupply: integer): string;
    function DestroyPiecesAndCountVPs(var Player1VPs: integer;
      var Player2VPs: integer): boolean;
    function GetGridAsString(P1Turn: boolean): string;
    function GetPieceTypeInTile(ID: integer): string;
  private
    function CheckTileIndexIsValid(TileToCheck: integer): boolean;
  …
    function GetFogOfWar(ID : integer) : boolean;
  end;

function HexGrid.GetFogOfWar(ID : integer) : boolean;
var
  ListOfNeighbours : TTileArray;
  ListToCheck : TTileArray;
  N : Tile;
  ThePiece : Piece;
  Index : integer;
  NeighbourIndex : integer;
begin
  ListOfNeighbours := Tiles[ID].GetNeighbours();
  ListToCheck := Tiles[ID].GetNeighbours();
  setlength(ListToCheck,length(ListToCheck) + 1);
  ListToCheck[high(ListToCheck)] := Tiles[ID];
  for Index := 0 to high(ListOfNeighbours) do
  begin
    N := ListOfNeighbours[Index];
    for NeighbourIndex := 0 to (length(N.GetNeighbours()) - 1) do
    begin
      SetLength(ListToCheck, length(ListToCheck) + 1);
      ListToCheck[High(ListToCheck)] :=
N.GetNeighbours()[NeighbourIndex];
    end;
  end;
  for Index := 0 to high(ListToCheck) do
  begin
    N := ListToCheck[Index];
    ThePiece := N.GetPieceInTile();
    if ThePiece <> nil then
      if ThePiece.GetBelongsToPlayer1() = Player1Turn then
      begin
        GetFogOfWar := False;
```

```
        exit;
      end;
  end;
  GetFogOfWar := True;
end;


function HexGrid.GetPieceTypeInTile(ID: integer): string;
var
  ThePiece: Piece;
begin
  ThePiece := Tiles[ID].GetPieceInTile();
  if (ThePiece = nil) then
    GetPieceTypeInTile := ' '
  else
  begin
    if GetFogOfWar(ID) then
      GetPieceTypeInTile := ' '
    else
      GetPieceTypeInTile := ThePiece.GetPieceType();
  end;
end;
```

JAVA

Question			Marks
07	1		12

```
Console.write("Enter value for n: ");
int number = Integer.parseInt(Console.readLine());
int numbersFoundSoFar = 0;
int currentNumber = 1;
int sumOfDigits;
while (numbersFoundSoFar != number) {
  int temp = currentNumber;
  sumOfDigits = 0;
  String numAsString = currentNumber + "";
    for (int count = 0; count < numAsString.length(); count++) {
        sumOfDigits +=
Integer.parseInt(numAsString.charAt(count)+"");
    }
    if (currentNumber % sumOfDigits == 0) {
        numbersFoundSoFar++;
        if (numbersFoundSoFar == number) {
            Console.writeLine(currentNumber);
        }
    }
    currentNumber++;
}
```

Alternative answer

```
Console.write("Enter value for n: ");
int number = Integer.parseInt(Console.readLine());
int numbersFoundSoFar = 0;
int currentNumber = 1;
int sumOfDigits;
while (numbersFoundSoFar != number) {
  int temp = currentNumber;
  sumOfDigits = 0;
  while (temp > 0) {
    sumOfDigits += temp % 10;
    temp /= 10;
  }
    if (currentNumber % sumOfDigits == 0) {
        numbersFoundSoFar++;
        if (numbersFoundSoFar == number) {
            Console.writeLine(currentNumber);
        }
    }
    currentNumber++;
}
```

Recursive answer

```
static int sumDigits(int num) {
    int sum;
    if (num == 0) {
        sum = 0;
    } else {
        sum = num % 10 + sumDigits(num/10);
    }
    return sum;
}

public static void main(String[] args) {

    Console.write("Enter value for n: ");
    int number = Integer.parseInt(Console.readLine());
    int nthHarshad = 0;
    int counter = 1;
    int value;
    while (number != nthHarshad) {
        value = sumDigits(counter);
        if (counter % value == 0) {
          nthHarshad += 1;
        }
        if (number == nthHarshad) {
          Console.writeLine(counter);
    }
        counter = counter + 1;
    }
}
```

12 **1** **5**

```
public Object[] destroyPiecesAndCountVPs(int player1VPs, int
player2VPs) {
    boolean baronDestroyed = false;
    List<Tile> listOfTilesContainingDestroyedPieces = new
ArrayList<>();
    for (Tile t : tiles) {
        if (t.getPieceInTile() != null) {
            List<Tile> listOfNeighbours = new
ArrayList<>(t.getNeighbours());
            int noOfConnections = 0;
            for (Tile n : listOfNeighbours) {
                if ( n.getPieceInTile() != null) {
                    noOfConnections += 1;
                }
            }
            Piece thePiece = t.getPieceInTile();
            if (noOfConnections >=
thePiece.getConnectionsNeededToDestroy()) {
                thePiece.destroyPiece();
                if
(thePiece.getPieceType().toUpperCase().equals("B")) {
                    baronDestroyed = true;
                }
                listOfTilesContainingDestroyedPieces.add(t);
                if (thePiece.getBelongsToplayer1()) {
                    player2VPs += thePiece.getVPs();
                } else {
                    player1VPs += thePiece.getVPs();
                }
```

```
            } else {

if(thePiece.getPieceType().toUpperCase().equals("L")) {
                    if (thePiece.getBelongsToplayer1()) {
                        player1VPs += 1;
                    } else {
                        player2VPs += 1;
                    }
                }
            }
        }
    }
    for (Tile t : listOfTilesContainingDestroyedPieces) {
        t.setPiece(null);
    }
    return new Object[]{baronDestroyed, player1VPs, player2VPs};
}
```

13 **1** **7**

```
    public void addPiece(boolean belongsToPlayer1, String
typeOfPiece, int location) {
        Piece newPiece;
        switch (typeOfPiece) {
            case "Baron":
                newPiece = new BaronPiece(belongsToPlayer1);
                break;
            case "LESS":
                newPiece = new LESSPiece(belongsToPlayer1);
                break;
            case "PBDS":
                newPiece = new PBDSPiece(belongsToPlayer1);
                break;
            case "Ranger":
                newPiece = new RangerPiece(belongsToPlayer1);
                break;
            default:
                newPiece = new Piece(belongsToPlayer1);
                break;
        }
        pieces.add(newPiece);
        tiles.get(location).setPiece(newPiece);
    }

class RangerPiece extends Piece {
    public RangerPiece(boolean player1) {
        super(player1);
        pieceType = "R";
    }

    @Override
    public int checkMoveIsValid(int distanceBetweenTiles, String
startTerrain, String endTerrain) {
        if (startTerrain.equals("#") &&
startTerrain.equals(endTerrain))
        {
            return fuelCostOfMove;
```

```
        }
        if (distanceBetweenTiles == 1) {
            if (startTerrain.equals("~") || endTerrain.equals("~"))
{
                return fuelCostOfMove * 2;
            } else {
                return fuelCostOfMove;
            }
        }
        return -1;
    }
}
```

14 **1**

```
boolean checkCommandIsValid(List<String> items) {
    if (items.size() > 0) {
    switch (items.get(0)) {
            case "move":
                return checkMoveCommandFormat(items);
            case "dig":
            case "saw":
            case "spawn":
                return checkStandardCommandFormat(items);
            case "upgrade":
                return checkUpgradeCommandFormat(items);
            case "burn":
                return true;
        }
    }
    return false;
}

    public Object[] executeCommand(List<String> items, int
fuelChange, int lumberChange,
                                   int supplyChange, int
fuelAvailable , int lumberAvailable,
                                   int piecesInSupply ) {
        int lumberCost;
        switch (items.get(0))
        {
            case "move":
                ...
            case "burn":
                if (lumberAvailable < 1) {
                    return new Object[] {"Cannot burn lumber ",
fuelChange, lumberChange, supplyChange};
                }
                Random rNoGen = new Random();
                int burnAmount = rNoGen.nextInt(lumberAvailable)+1;
                lumberChange -= burnAmount;
                fuelChange += burnAmount;

        }
```

8

```
        return new Object[] {"Command executed", fuelChange,
lumberChange, supplyChange};
    }
```

15 **1**

```
public boolean getFogOfWar(int index) {
    List<Tile> listOfNeighbours = tiles.get(index).getNeighbours();
    List<Tile> listToCheck = new ArrayList<>();
    listToCheck.add(tiles.get(index));
    for (Tile n : listOfNeighbours) {
        listToCheck.addAll(n.getNeighbours());
    }
    for (Tile n : listToCheck) {
        Piece thePiece = n.getPieceInTile();
        if (thePiece != null) {
            if (thePiece.getBelongsToPlayer1() == player1Turn) {
                return false;
            }
        }
    }
    return true;
}

public String getPieceTypeInTile(int id) {
    Piece thePiece = tiles.get(id).getPieceInTile();
    if (thePiece == null) {
        return " ";
    } else {
        if (getFogOfWar(id)) {
            return " ";
        }
        return thePiece.getPieceType();
    }
}
```

13