



AS COMPUTER SCIENCE 7516/1

Paper 1

Mark scheme

June 2025

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.

No student should be disadvantaged on the basis of their gender identity and/or how they refer to the gender identity of others in their exam responses.

A consistent use of 'they/them' as a singular and pronouns beyond 'she/her' or 'he/him' will be credited in exam responses in line with existing mark scheme criteria.

Further copies of this mark scheme are available from [aqa.org.uk](https://www.aqa.org.uk)

Copyright information

AQA retains the copyright on all its publications. However, registered schools/colleges for AQA are permitted to copy material from this booklet for their own internal use, with the following important exception: AQA cannot give permission to schools/colleges to photocopy any material that is acknowledged to a third party even for internal use within the centre.

Copyright © 2025 AQA and its licensors. All rights reserved.

AS Computer Science

Paper 1 (7516/1) – applicable to all programming languages A, B, D and E

June 2025

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.** - 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.

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.

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 **16.1**, the marks available for the AO3 elements are as follows:

AO3 (design)	2 marks
AO3 (programming)	8 marks

In question **17.1**, the marks available for the AO3 elements are as follows:

AO3 (design)	3 marks
AO3 (programming)	9 marks

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

Section A

Qu	Marks																																												
01	3 marks for AO2 (application) <table><tr><th>M</th><th>i</th><th>Symbol</th><th>Current</th><th>Output</th></tr><tr><td>"1001"</td><td></td><td></td><td>0</td><td></td></tr><tr><td></td><td>0</td><td>1</td><td>20</td><td></td></tr><tr><td></td><td>1</td><td>0</td><td>14</td><td></td></tr><tr><td></td><td>2</td><td>0</td><td>4</td><td></td></tr><tr><td></td><td>3</td><td>1</td><td>24</td><td>X</td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr></table> 1 mark for each correct set of values in the correct sequence (boxed in red) I. presence/absence of quotation marks around values in all columns and case in Output column Max 2 if any errors				M	i	Symbol	Current	Output	"1001"			0			0	1	20			1	0	14			2	0	4			3	1	24	X											3
M	i	Symbol	Current	Output																																									
"1001"			0																																										
	0	1	20																																										
	1	0	14																																										
	2	0	4																																										
	3	1	24	X																																									
02	2 marks for AO1 (understanding) (Code within) WHILE/Figure 3 loop structure is not executed // final value of X remains the same; (Code within) REPEAT/Figure 4 loop structure will be executed (once) // the value of X changes/decreases; The WHILE loop tests the condition at the start of the loop whereas the REPEAT loop tests the condition at the end of the loop; Max 2				2																																								

03	1	8 marks for AO3 (programming) Mark as follows: 1. Correct variable declarations for S, Max, Matched, i, Letter1, Letter2; I. case Note to examiners: If a language allows variables to be used without explicit declaration, (eg Python), then this mark should be awarded if the correct variables exist in the program code and the first value they are assigned is of the correct data type. 2. Correct WHILE loop syntax allowed by the programming language and correct condition; 3. Correct prompt "Enter a word or phrase: " and S assigned value entered by user; I. case 4. Correct calculation of Max; 5. FOR loop iterates correct number of times; 6. Correct IF THEN statement syntax allowed by the programming language and correct condition within FOR loop; R. if inappropriate values for Letter1 or Letter2 7. Correct assignment to Matched in THEN part; 8. Correct IF THEN ELSE statement syntax allowed by the programming language and correct condition after FOR loop and correct output; I. case, spelling, spacing Max 7 if code does not function correctly	8
03	2	Mark is for AO3 (evaluate) **** SCREEN CAPTURE **** Must match code from 03.1. Code for 03.1 must be sensible. Screen capture showing: <pre>Enter a word or phrase: madam Palindrome Enter a word or phrase: maam Palindrome Enter a word or phrase: adam Not a palindrome Enter a word or phrase: aam Not a palindrome Enter a word or phrase: x Palindrome</pre>	1

03	3	Mark is for AO3 (evaluate) Algorithm makes unnecessary comparisons // by example: eg first and last letters compared twice; The loop continues to iterate after it has been identified that a word is not a palindrome; The loop continues to iterate after <code>Matched</code> has been set to <code>False</code> ; Max 1	1
----	---	---	---

04	1	Mark is for AO1 (knowledge) A named/callable (out of line) block of code (that may be executed by writing the name in a program statement);	1
04	2	3 marks for AO1 (understanding) Easier to re-use code; Easier to understand; Easier to debug/update/maintain/test; Easier to develop a solution // supports structured approach; Less code // faster to develop a solution; Facilitates multiple programmers working on a program simultaneously; Reduces/eliminates side-effects; A. enables use of <u>local</u> variables, which only use memory when subroutine is executing Max 3	3
04	3	2 marks for AO1 (understanding) Avoids the use of global variables // makes subroutines self-contained/encapsulated; Makes it easier to use the subroutine with different values/expressions/variables; Makes it easier to reuse the subroutine in a different program; Makes it easier to test the subroutine independently of the rest of the program; Makes it clearer which values from outside the subroutine are being used inside the subroutine (as they are explicitly listed); Max 2	2

05		Mark is for AO1 (understanding) (The detail of) how the data are (actually) represented is hidden; New kinds of data objects/structures can be constructed from previously defined types (of data objects); By example (eg stack/queue/tree implemented as an array); Max 1	1
----	--	---	---

Section B

Qu	Marks	
06	Mark is for AO2 (analyse) Found/ObstacleFound/ValidDirection/ValidDistance /ObstacleInPath; A. TreasureFound R. if any additional code R. if spelt incorrectly I. case and spacing	1
07	Mark is for AO2 (analyse) FindLandingPlace; R. if any additional code R. if spelt incorrectly I. case and spacing	1
08	1 Mark is for AO2 (analyse) CheckDistance; R. if any additional code R. if spelt incorrectly I. case and spacing	1
08	2 2 marks for AO2 (analyse) The string/character supplied may not be (convertible to) an integer; Without exception handling the program would crash;	2

09	1	Mark is for AO2 (analyse) Pirate; A. PirateRecord Java only: MoveCheckRecord; R. if any additional code R. if spelt incorrectly I. case and spacing	1
09	2	Mark is for AO2 (analyse) Map/HiddenMap/MapSize; A. MapSizeRecord R. if any additional code R. if spelt incorrectly I. case and spacing	1

10	1	Mark is for AO1 (knowledge) Subroutine/procedure/function/method; A. module/block of code	1
10	2	Mark is for AO2 (analyse) PirateWalks/PirateDigs; I. case and spacing	1
10	3	Mark is for AO2 (analyse) PirateDigs/PirateWalks; R. if given in 10.2 I. case and spacing	1
10	4	Mark is for AO1 (understanding) (During the) design (stage); R. more than one stage given	1

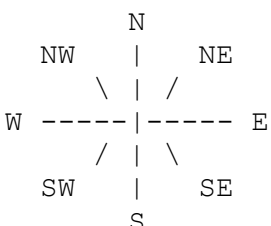
11		2 marks for AO2 (analyse) Without this test the X would be overwritten / replaced by sand; ... after the pirate's/player's (first) move; A. ... at the beginning of the game Max 2	2
-----------	--	--	----------

12		3 marks for AO2 (analyse) <i>Similarity (Max 1):</i> Both loops count up; Both loops have a step of one; Both loops use <code>Column</code>; <i>Differences (Max 2):</i> The loop for "E" starts at the start column whereas the loop for "W" starts at the end column; The loop for "E" ends at the end column whereas the loop for "W" ends at the start column; The start and end values for the loops are reversed (between E and W); I. Exact start/end points as long as reference is made to the start and end columns	3
-----------	--	--	----------

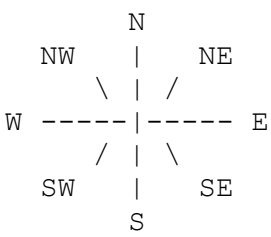
13		3 marks for AO2 (analyse) <code>Direction/NumberOfSquares/distance</code> might be valid (independently); But the combination of these ...; ... could result in <code>Row</code> and/or <code>Column</code> to be outside the boundaries of the map; (<code>CheckPath</code>) would attempt to address elements beyond the bounds of the (<code>Map</code>) data structure // would then malfunction/crash; Max 3	3
-----------	--	---	----------

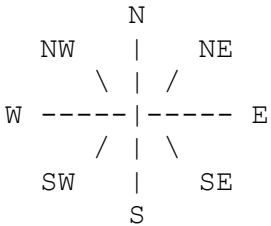
14	1	2 marks for AO2 (analyse) User presses Enter (without entering any other value first) when asked for pirate action; The pirate finds the treasure;	2
14	2	Mark is for AO2 (analyse) Finding a coconut/treasure;	1
14	3	Mark is for AO2 (analyse) Pirate walking/digging;	1

Section C

Qu		Marks	
15	1	4 marks for AO3 (programming) Marking guidance: Evidence of AO3 programming – 4 marks: Evidence of programming to look for in response: 1. Field initialised in <code>ResetPirateRecord</code>; 2. Selection structure to detect all eight compass points; 3. Correct calculation of hours walked for all eight directions; 4. In <code>DisplayResults</code> output number of hours walked with suitable message; Max 3 if any errors	4
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. Screen capture showing: <pre> 012345678901234567890123456789012345678901 0 WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW 1 WWWWWW.....WWWWWWWWWW 2 WWWWW.....WWWWWWWW 3 WWW.....WWWWWW 4 WW.....R.....WWW 5 W.....WWW 6 W.....WW 7 W.....*.*.*.....W 8 W.....W 9 W.....*.....*.....H.....W 0 W.....*.....P.....W 1 W.....*.....*.....W 2 WW.....BBWW 3 WWW.....BBWWW 4 WWWWW.....BBBBBB...BBWWW 5 WWWWW.....BBWWWWBBBWWWWWW 6 WWWWWWW.....BBBWWWWWWWWWWWWWW 7 WWWWWWWWW.....BBWWWWWWWWWWWWWWWW 8 WWWWWWWWWWW.....X...BBWWWWWWWWWWWWWWWW 9 WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW </pre>  Pirate to walk (W) or dig (D), to finish game press Enter: 0.0 hours spent digging 20.2 hours spent walking The score is 90 Press Enter to finish	1

16	1	2 marks for AO3 (design) and 8 marks for AO3 (programming) Marking guidance: Evidence of AO3 design – 2 marks: Evidence of design to look for in response: 1. Identify the need to ask for row and column; 2. Identify the need to test for sand or water to check for a beach; Note: 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 of AO3 programming – 8 marks: Evidence of programming to look for in response: 3. Ask for user input whether different landing place required; 4. Correctly assign user input to entered row and column variables; 5. Check that the entered row and column represent sand; 6. Correctly check that entered position is next to water; 7. only if on sand and next to water ... 8. ... Set pirate position to entered position; 9. ... and put pirate on map and display map; 10. ... otherwise output a suitable message; Max 9 if any errors	10
----	---	--	----

16	2	Mark is for AO3 (evaluate) **** SCREEN CAPTURE **** Must match code from 16.1, including prompts on screen capture matching those in code. Code for 16.1 must be sensible. X marks the spot where the pirate comes ashore Do you want the pirate to land elsewhere? (Y/N): Y Which row is the pirate coming ashore? 6 Which column is the pirate coming ashore? 1 <pre>012345678901234567890123456789012345678901 0 WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW 1 WWWWWW.....WWWWWWWWWW 2 WWWWW.....WWWWWWWW 3 WWW.....WWWWWW 4 WW.....R.....WWW 5 W.....WWW 6 WP.....WW 7 W.....*.*.*.....W 8 W.....W 9 W.....*.H.....W 0 W.....*.....W 1 W.....*.....W 2 WW.....BBWW 3 WWW.....BBWWW 4 WWWWW.....BBBBB...BBWWW 5 WWWWWW.....BBWWWWBBBWWWWW 6 WWWWWW.....BBBWWWWWWWWWWWWWW 7 WWWWWWWWW.....BBWWWWWWWWWWWWWW 8 WWWWWWWWWWW.....X.....BBWWWWWWWWWWWWWW 9 WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW</pre> 	1
----	---	--	---

16	3	Mark is for AO3 (evaluate) **** SCREEN CAPTURE **** Must match code from 16.1, including prompts on screen capture matching those in code. Code for 16.1 must be sensible. X marks the spot where the pirate comes ashore Do you want the pirate to land elsewhere? (Y/N): Y Which row is the pirate coming ashore? 17 Which column is the pirate coming ashore? 21 That was not a good landing place, so the pirate lands at X Pirate to walk (W) or dig (D), to finish game press Enter: W Enter length (between 1 and 9) and direction (N, NE, E, SE, S, SW, W, NW): 1N <pre>012345678901234567890123456789012345678901 0 WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW 1 WWWWWW.....WWWWWWWWWW 2 WWWWW.....WWWWWWWW 3 WWW.....WWWWWW 4 WW.....R.....WWW 5 W.....WWW 6 W.....WW 7 W.....*.*.*.....W 8 W.....W 9 W.....*.H.....W 0 W.....*.....W 1 W.....*.....W 2 WW.....BBWW 3 WWW.....BBWWW 4 WWWWW.....BBBBB...BBWWW 5 WWWWWW.....BBWWWWBBBWWWWWW 6 WWWWWW.....BBBWWWWWWWWWWWWWW 7 WWWWWWWWW.....P.....BBWWWWWWWWWWWWWWWW 8 WWWWWWWWWWW.....X....BBWWWWWWWWWWWWWWWW 9 WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW</pre> 	1
----	---	---	---

17	1	3 marks for AO3 (design) and 9 marks for AO3 (programming)	12												
<table><tr><th>Level</th><th>Description</th><th>Mark Range</th></tr><tr><td>3</td><td>A line of reasoning has been followed to arrive at a logically structured working or almost fully working programmed solution. All of the appropriate design decisions have been taken.</td><td>9–12</td></tr><tr><td>2</td><td>There is evidence that a line of reasoning has been partially followed. There is evidence of some appropriate design work. This is a partially working programmed solution.</td><td>5–8</td></tr><tr><td>1</td><td>An attempt has been made to amend the subroutine <code>PirateWalks</code> or to create one of the other two subroutines. Some appropriate programming statements have been written. There is little evidence to suggest that a line of reasoning has been followed or that the solution has been designed. The statements written may or may not be syntactically correct and the subroutines will have very little or none of the extra required functionality. It is unlikely that any of the key design elements of the task have been recognised.</td><td>1–4</td></tr></table>				Level	Description	Mark Range	3	A line of reasoning has been followed to arrive at a logically structured working or almost fully working programmed solution. All of the appropriate design decisions have been taken.	9–12	2	There is evidence that a line of reasoning has been partially followed. There is evidence of some appropriate design work. This is a partially working programmed solution.	5–8	1	An attempt has been made to amend the subroutine <code>PirateWalks</code> or to create one of the other two subroutines. Some appropriate programming statements have been written. There is little evidence to suggest that a line of reasoning has been followed or that the solution has been designed. The statements written may or may not be syntactically correct and the subroutines will have very little or none of the extra required functionality. It is unlikely that any of the key design elements of the task have been recognised.	1–4
Level	Description	Mark Range													
3	A line of reasoning has been followed to arrive at a logically structured working or almost fully working programmed solution. All of the appropriate design decisions have been taken.	9–12													
2	There is evidence that a line of reasoning has been partially followed. There is evidence of some appropriate design work. This is a partially working programmed solution.	5–8													
1	An attempt has been made to amend the subroutine <code>PirateWalks</code> or to create one of the other two subroutines. Some appropriate programming statements have been written. There is little evidence to suggest that a line of reasoning has been followed or that the solution has been designed. The statements written may or may not be syntactically correct and the subroutines will have very little or none of the extra required functionality. It is unlikely that any of the key design elements of the task have been recognised.	1–4													
Marking guidance: Evidence of AO3 design – 3 marks: Evidence of design to look for in response: 1. Attempt to test for pirate west of hut as a subroutine. 2. Recognise the need for a nested loop to find the rock/treasure in the hidden map. 3. Attempt to calculate distance of treasure from rock in one direction. Note: 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 of AO3 programming – 9 marks: Evidence of programming to look for in response: 4. Correct parameters and return values for <code>WestOfHut</code>. 5. Correct parameters for <code>GetClue</code>. 6. Correctly search for rock. 7. Correctly search for treasure. 8. Correctly calculate distance of treasure from rock in NS and EW direction within subroutine <code>GetClue</code> 9. Output distance in NS and EW direction within subroutine <code>GetClue</code> DPT if in wrong subroutine 10. Ensure distances are positive. 11. Call <code>WestOfHut</code> in correct place in <code>PirateWalks</code>. 12. Call <code>GetClue</code> under correct conditions. Max 11 if code does not function correctly															

17	2	Mark is for AO3 (evaluate) **** SCREEN CAPTURE **** Must match code from 17.1, including prompts on screen capture matching those in code. Code for 17.1 must be sensible. Screen capture showing: <pre> 012345678901234567890123456789012345678901 0 WW 1 WWWWWW.....WWWWWWWWWWWW 2 WWWWW.....WWWWWWWWWW 3 WWWW.....WWWWWW 4 WW.....R.....WWW 5 W.....WWW 6 W.....WW 7 W.....*.*.*.....W 8 W.....W 9 W.....*.PH.....W 0 W.....*.....W 1 W.....*.....W 2 WW.....BBWW 3 WWW.....BBWWW 4 WWWWW.....BBBBBB...BBWWW 5 WWWWWW.....BBWWWWWWBBBBBWWWWW 6 WWWWWW.....BBBWWWWWWWWWWWWWWWW 7 WWWWWWWWWWW.....BBWWWWWWWWWWWWWWWW 8 WWWWWWWWWWW.....X....BBWWWWWWWWWWWWWWWW 9 WW </pre> The treasure is 2 squares away from the rock in the North - South direction The treasure is 4 squares away from the rock in the East - West direction	1
----	---	---	---

VB.Net

03	1	<pre>Sub Main() Dim S As String = "" Dim Letter1, Letter2 As Char Dim Max As Integer Dim Matched As Boolean ' MP1 While S <> "x" ' MP2 Console.WriteLine("Enter a word or phrase: ") S = Console.ReadLine() ' MP3 Max = S.Length - 1 ' MP4 Matched = True For i = 0 To Max ' MP5 Letter1 = S(i) Letter2 = S(Max - i) If Letter1 <> Letter2 Then ' MP6 Matched = False ' MP7 End If Next If Matched = True Then Console.WriteLine("Palindrome") Else Console.WriteLine("Not a palindrome") ' MP8 End If End While Console.ReadLine() End Sub</pre>	8
----	---	---	---

15	1	<pre> Sub ResetPirateRecord(ByRef Pirate As PirateRecord) Pirate.Row = 0 Pirate.Column = 0 Pirate.Score = 100 Pirate.DigTime = 0.0 Pirate.TreasureFound = False Pirate.NumberOfCoinsFound = 0 Pirate.WalkTime = 0.0 ' MP1 Task 1 End Sub Sub PirateWalks(Map(,) As String, MapSize As MapSizeRecord, HiddenMap(,) As String, ByRef Pirate As PirateRecord) Dim ObstacleInPath As Boolean = True Dim ValidDistance As Boolean = False Dim ValidDirection As Boolean = False Dim WalkData, Direction As String Dim Row, Column, NumberOfSquares As Integer While ObstacleInPath Or Not ValidDistance Or Not ValidDirection Console.WriteLine("Enter length (1 to 9) and direction (N, NE, E, SE, S, SW, W, NW): ") WalkData = Console.ReadLine() Row = Pirate.Row Column = Pirate.Column ValidDistance = CheckDistance(WalkData(0), NumberOfSquares) Direction = WalkData.Substring(1) ValidDirection = CheckDirection(Direction, Row, Column, NumberOfSquares) If Row >= MapSize.Rows Or Column >= MapSize.Columns Or Row < 0 Or Column < 0 Then ValidDirection = False Console.WriteLine("Error") End If If ValidDirection Then ObstacleInPath = CheckPath(Map, Pirate.Row, Pirate.Column, Row, Column, Direction) If ObstacleInPath Then Console.WriteLine("Pirate can't walk this way as there is an obstacle in the way") End If End If End While Move(Map, MapSize, Pirate, Row, Column) If Len(Direction) = 1 Then Pirate.WalkTime += NumberOfSquares ' MP2 Else Pirate.WalkTime += NumberOfSquares * 1.4 ' MP3 Task 2 End If End Sub Sub DisplayResults(Pirate As PirateRecord) If Pirate.NumberOfCoinsFound > 0 Then Console.WriteLine(\$"{Pirate.NumberOfCoinsFound} gold coins found") End If Console.WriteLine(\$"{Pirate.DigTime:N} hours spent digging") Console.WriteLine(\$"{Pirate.WalkTime:N} hours spent walking") ' MP4 Task3 Console.WriteLine(\$"The score is { Pirate.Score }") End Sub </pre>	4
----	---	--	---

	Alternative answer 1 for mark points 2 and 3: <pre> Move(Map, MapSize, Pirate, Row, Column) If "EWNS".Contains(Direction) Then 'Task 2 NB EW or WE needs to be first Pirate.WalkTime += NumberOfSquares ' MP2 Else Pirate.WalkTime += NumberOfSquares * 1.4 ' MP3 Task 2 End If </pre> Alternative answer 2 for mark points 2 and 3: <pre> Move(Map, MapSize, Pirate, Row, Column) If {"N", "S", "E", "W"}.Contains(Direction) Then Pirate.WalkTime += NumberOfSquares ' MP2 Else Pirate.WalkTime += NumberOfSquares * 1.4 ' MP3 Task 2 End If </pre> Alternative answer 3 for mark points 2 and 3: <pre> Move(Map, MapSize, Pirate, Row, Column) If Direction = "N" Or Direction = "S" Or Direction = "E" Or Direction = "W" Then 'Task 2 Pirate.WalkTime += NumberOfSquares ' MP2 Else Pirate.WalkTime += NumberOfSquares * 1.4 ' MP3 Task 2 End If </pre>	
--	--	--

16	1	<pre> Sub FindLandingPlace(Map(,) As String, MapSize As MapSizeRecord, ByRef Pirate As PirateRecord) Dim Found As Boolean = False Dim Row, Column As Integer Row = 0 While Not Found And Row < MapSize.Rows Column = 0 While Not Found And Column < MapSize.Columns If Map(Row, Column) = "X" Then Found = True Pirate.Row = Row Pirate.Column = Column End If Column += 1 End While Row += 1 End While DisplayMap(Map, MapSize) Console.WriteLine("X marks the spot where the pirate comes ashore") Console.WriteLine() Console.Write("Do you want the pirate to land elsewhere? (Y/N) ") Dim Reply As String = Console.ReadLine().ToUpper ' MP3 If Reply = "Y" Then Console.Write("Which row is the pirate coming ashore? ") Row = Convert.ToInt32(Console.ReadLine()) Console.Write("Which column is the pirate coming ashore? ") ' MP1 Column = Convert.ToInt32(Console.ReadLine()) ' MP4 If Map(Row, Column) = SAND And (Map(Row + 1, Column) = WATER Or Map(Row - 1, Column) = WATER Or Map(Row, Column + 1) = WATER Or Map(Row, Column - 1) = WATER) Then ' MP2 MP5 MP6 MP7 Pirate.Row = Row Pirate.Column = Column ' MP8 Map(Row, Column) = PIRATES DisplayMap(Map, MapSize) ' MP9 Else Console.WriteLine("That was not a good landing place, so the pirate lands at X") ' MP10 End If End If End Sub </pre>	10
----	---	--	----

17	1	<pre> Function WestOfHut(Map(,) As String, Pirate As PirateRecord) As Boolean If Map(Pirate.Row, Pirate.Column + 1) = HUT Then ' MP1 Task 1 Return True Else Return False ' MP4 End If End Function Sub GetClue(Map(,) As String, MapSize As MapSizeRecord, HiddenMap(,) As String) ' MP5 Task 2 Dim RockRow, RockColumn, TreasureRow, TreasureColumn as Integer Dim NSDistance, EWDistance As Integer For Row = 0 To MapSize.Rows - 1 For Column = 0 To MapSize.Columns - 1 ' MP2 If Map(Row, Column) = ROCK Then ' MP6 RockRow = Row RockColumn = Column End If If HiddenMap(Row, Column) = TREASURE Then ' MP7 TreasureRow = Row TreasureColumn = Column End If Next Next NSDistance = Math.Abs(RockRow - TreasureRow) ' MP3 EWDistance = Math.Abs(RockColumn - TreasureColumn) ' MP8 MP10 Console.WriteLine(\$"The treasure is {NSDistance} squares way from the rock in the North - South direction") Console.WriteLine(\$" The treasure is {EWDistance} squares away from the rock in the East - West direction") ' MP9 End Sub Sub PirateWalks(Map(,) As String, MapSize As MapSizeRecord, HiddenMap(,) As String, ByRef Pirate As PirateRecord) Dim ObstacleInPath As Boolean = True Dim ValidDistance As Boolean = False Dim ValidDirection As Boolean = False Dim WalkData, Direction As String Dim Row, Column, NumberOfSquares As Integer While ObstacleInPath Or Not ValidDistance Or Not ValidDirection Console.Write("Enter length (1 to 9) and direction (N, NE, E, SE, S, SW, W, NW): ") WalkData = Console.ReadLine() Row = Pirate.Row Column = Pirate.Column ValidDistance = CheckDistance(WalkData(0), NumberOfSquares) Direction = WalkData.Substring(1) ValidDirection = CheckDirection(Direction, Row, Column, NumberOfSquares) If Row >= MapSize.Rows Or Column >= MapSize.Columns Or Row < 0 Or Column < 0 Then ValidDirection = False Console.WriteLine("Error") End If If ValidDirection Then ObstacleInPath = CheckPath(Map, Pirate.Row, Pirate.Column, Row, Column, Direction) If ObstacleInPath Then Console.WriteLine("Pirate can't walk this way as there is an obstacle in the way") End If End If End While End Sub </pre>	12
----	---	--	----

		<pre> End If End While Move(Map, MapSize, Pirate, Row, Column) ... If WestOfHut(Map, Pirate) Then ' MP11 Task 3 GetClue(Map, MapSize, HiddenMap) ' MP12 End If End Sub </pre>	
--	--	---	--

Python 3

03	1	<pre> S = "" while S != "x": # MP1 # MP2 S = input("Enter a word or phrase: ") # MP3 Max = len(S) - 1 # MP4 Matched = True for i in range(Max + 1): # MP5 Letter1 = S[i] Letter2 = S[Max - i] if Letter1 != Letter2: # MP6 Matched = False # MP7 if Matched: print("Palindrome") else: print("Not a palindrome") # MP8 </pre>	8
----	---	---	---

15	1	<pre> def ResetPirateRecord(Pirate): Pirate.Row = 0 Pirate.Column = 0 Pirate.Score = 100 Pirate.DigTime = 0.0 Pirate.TreasureFound = False Pirate.NumberOfCoinsFound = 0 Pirate.WalkTime = 0.0 # MP1 Task 1 def PirateWalks(Map, MapSize, HiddenMap, Pirate): ObstacleInPath = True ValidDistance = False ValidDirection = False while ObstacleInPath or not ValidDistance or not ValidDirection: WalkData = input("Enter length (1 to 9) and direction (N, NE, E, SE, S, SW, W, NW): ") Row = Pirate.Row Column = Pirate.Column ValidDistance, NumberOfSquares = CheckDistance(WalkData[0]) Direction = WalkData[1:] ValidDirection, Row, Column = CheckDirection(Direction, Row, Column, NumberOfSquares) if Row >= MapSize.Rows or Column >= MapSize.Columns or Row < 0 or Column < 0: ValidDirection = False print("Error") if ValidDirection: ObstacleInPath = CheckPath(Map, Pirate.Row, Pirate.Column, Row, Column, Direction) if ObstacleInPath: print("Pirate can't walk this way as there is an obstacle in the way") Move(Map, MapSize, Pirate, Row, Column) if len(Direction) == 1: Pirate.WalkTime += NumberOfSquares # Task 2 # MP2 else: # Pirate.WalkTime += 1.4 * NumberOfSquares # MP3 def DisplayResults(Pirate): if Pirate.NumberOfCoinsFound > 0: print(f"{Pirate.NumberOfCoinsFound} gold coins found") print(f"{Pirate.DigTime} hours spent digging") print(f"{Pirate.WalkTime} hours spent walking") # MP4 Task 3 print(f"The score is {Pirate.Score}") Alternative answer 1 for mark points 2 and 3: Move(Map, MapSize, Pirate, Row, Column) if Direction in "WESN": Pirate.WalkTime += NumberOfSquares # Task 2 # MP2 else: # Pirate.WalkTime += 1.4 * NumberOfSquares # MP3 Alternative answer 2 for mark points 2 and 3: Move(Map, MapSize, Pirate, Row, Column) if Direction in ["N", "S", "E", "W"]: Pirate.WalkTime += NumberOfSquares # Task 2 # MP2 else: # Pirate.WalkTime += 1.4 * NumberOfSquares # MP3 </pre>	4
----	---	---	---

		Alternative answer 3 for mark points 2 and 3: <pre> Move(Map, MapSize, Pirate, Row, Column) if Direction == "N" or Direction == "S" or Direction == "E" or Direction == "W": Pirate.WalkTime += NumberOfSquares else: Pirate.WalkTime += 1.4 * NumberOfSquares </pre> # Task 2 # MP2 # MP3	
--	--	---	--

16	1	<pre> def FindLandingPlace(Map, MapSize, Pirate): Found = False Row = 0 while not Found and Row < MapSize.Rows: Column = 0 while not Found and Column < MapSize.Columns: if Map[Row][Column] == 'X': Found = True Pirate.Row = Row Pirate.Column = Column Column += 1 Row += 1 DisplayMap(Map, MapSize) print("X marks the spot where the pirate comes ashore") print() Answer = input("Do you want the pirate to land elsewhere? (Y/N): ") if Answer == "Y": Row = 0 Column = 0 Row = int(input("Which row is the pirate coming ashore? ")) Column = int(input("Which column is the pirate coming ashore? ")) OnSand = False NextToWater = False if Map[Row][Column] == SAND: OnSand = True if Map[Row - 1][Column] == WATER or Map[Row + 1][Column] == WATER: NextToWater = True if Map[Row][Column + 1] == WATER or Map[Row][Column - 1] == WATER: NextToWater = True if OnSand and NextToWater: Pirate.Row = Row Pirate.Column = Column Map[Row][Column] = PIRATES DisplayMap(Map, MapSize) else: print("That was not a good landing place, so the pirate lands at X") </pre> # MP3 # MP1 # MP4 # MP5 # MP2 # MP6 # MP7 # MP8 # MP9 # MP10	10
----	---	---	----

17	1	<pre> def WestOfHut(Map, Pirate): if Map[Pirate.Row][Pirate.Column + 1] == HUT: # MP1 Task 1 return True # else: # return False # MP4 def GetClue(Map, MapSize, HiddenMap): # MP5 Task 2 for Row in range(MapSize.Rows): for Column in range(MapSize.Columns): # MP2 if Map[Row][Column] == ROCK: # MP6 RockRow = Row RockColumn = Column for Row in range(MapSize.Rows): for Column in range(MapSize.Columns): if HiddenMap[Row][Column] == TREASURE: # MP7 TreasureRow = Row TreasureColumn = Column DistanceFromRock_NS = TreasureRow - RockRow # MP3 if DistanceFromRock_NS < 0: DistanceFromRock_NS = - DistanceFromRock_NS DistanceFromRock_WE = TreasureColumn - RockColumn # MP8 if DistanceFromRock_WE < 0: DistanceFromRock_WE = - DistanceFromRock_WE # MP10 print(f"The treasure is {DistanceFromRock_NS} squares away from the rock in the North - South direction") # print(f"The treasure is {DistanceFromRock_WE} squares away from the rock in the East - West direction") # MP9 def PirateWalks(Map, MapSize, HiddenMap, Pirate): ObstacleInPath = True ValidDistance = False ValidDirection = False while ObstacleInPath or not ValidDistance or not ValidDirection: WalkData = input("Enter length (1 to 9) and direction (N, NE, E, SE, S, SW, W, NW): ") Row = Pirate.Row Column = Pirate.Column ValidDistance, NumberOfSquares = CheckDistance(WalkData[0]) Direction = WalkData[1:] ValidDirection, Row, Column = CheckDirection(Direction, Row, Column, NumberOfSquares) if Row >= MapSize.Rows or Column >= MapSize.Columns or Row < 0 or Column < 0: ValidDirection = False print("Error") if ValidDirection: ObstacleInPath = CheckPath(Map, Pirate.Row, Pirate.Column, Row, Column, Direction) if ObstacleInPath: print("Pirate can't walk this way as there is an obstacle in the way") Move(Map, MapSize, Pirate, Row, Column) ... if WestOfHut(Map, Pirate): # MP11 Task 3 GetClue(Map, MapSize, HiddenMap) # MP12 </pre>	12
----	---	---	----

C#

03	1	<pre> string s = ""; int max = 0; bool matched = true; string letter1 = ""; string letter2 = ""; // MP1 while (s != "x") // MP2 { Console.Write("Enter a word or phrase: "); s = Console.ReadLine(); // MP3 max = s.Length - 1; // MP4 matched = true; for (int i = 0; i <= max; i++) // MP5 { letter1 = s[i].ToString(); letter2 = s[max - i].ToString(); if (letter1 != letter2) // MP6 { matched = false; // MP7 } } if (matched == true) { Console.WriteLine("Palindrome"); } else { Console.WriteLine("Not a Palindrome"); // MP8 } } </pre>	8
-----------	----------	--	----------

15	1	<pre> public static void ResetPirateRecord(ref PirateRecord pirate) { pirate.row = 0; pirate.column = 0; pirate.score = 100; pirate.digTime = 0.0; pirate.treasureFound = false; pirate.numberOfCoinsFound = 0; pirate.walkTime = 0.0; // MP1 Task 1 } static void PirateWalks(string[,] map, MapSizeRecord mapSize, string[,] hiddenMap, ref PirateRecord pirate) { bool obstacleInPath; bool validDistance; bool validDirection; string walkData; string direction = ""; int row = 0; int column = 0; int numberOfSquares = 0; obstacleInPath = true; validDistance = false; validDirection = false; while (obstacleInPath !validDistance !validDirection) { Console.WriteLine("Enter length (1 to 9) and direction (N, NE, E, SE, S, SW, W, NW): "); walkData = Console.ReadLine(); row = pirate.row; column = pirate.column; (validDistance, numberOfSquares) = CheckDistance(walkData[0].ToString()); direction = walkData.Substring(1); (validDirection, row, column) = CheckDirection(direction, row, column, numberOfSquares); if (row >= mapSize.rows column >= mapSize.columns row < 0 column < 0) { validDirection = false; Console.WriteLine("Error"); } if (validDirection) { obstacleInPath = CheckPath(map, pirate.row, pirate.column, row, column, direction); if (obstacleInPath) { Console.WriteLine("Pirate can't walk this way as there is an obstacle in the way"); } } } Move(map, mapSize, ref pirate, row, column); if (direction.Length == 1) // Task 2 </pre>	4
----	---	---	---

		<pre>{ pirate.walkTime += numberOfSquares; // MP2 } else { pirate.walkTime += 1.4 * numberOfSquares; // MP3 } }</pre>	
--	--	---	--

16	1	<pre> public static void FindLandingPlace(string[,] map, MapSizeRecord mapSize, ref PirateRecord pirate) { bool found = false; int row = 0; while (!found && (row < mapSize.rows)) { int column = 0; while (!found && (column < mapSize.columns)) { if (map[row, column] == "X") { found = true; pirate.row = row; pirate.column = column; } column++; } row++; } DisplayMap(map, mapSize); Console.WriteLine("X marks the spot where the pirate comes ashore"); Console.WriteLine(); Console.Write("Do you want the pirate to land elsewhere (Y/N) "); string answer = Console.ReadLine(); // MP3 if (answer == "Y") { Console.Write("Which row is the pirate coming ashore? "); row = Convert.ToInt32(Console.ReadLine()); // MP1 Console.Write("Which column is the pirate coming ashore? "); int column = Convert.ToInt32(Console.ReadLine()); // MP4 bool onSand = false; bool nextToWater = false; if (map[row, column] == SAND) //MP5 { onSand = true; } if (map[row - 1, column] == WATER map[row + 1, column] == WATER) // MP2 { nextToWater = true; } if (map[row, column - 1] == WATER map[row, column + 1] == WATER) // MP6 { nextToWater = true; } if (onSand && nextToWater) // MP7 { pirate.row = row; pirate.column = column; // MP8 map[row, column] = PIRATES; DisplayMap(map, mapSize); // MP9 </pre>	10
----	---	--	----

		<pre> } else { Console.WriteLine("That was not a good landing place, so the pirate lands at X"); // MP10 } }</pre>	
--	--	---	--

17	1	<pre> private static bool WestOfHut(string[,] map, PirateRecord pirate) { if (map[pirate.row, pirate.column + 1] == HUT) // MP1 Task 1 { return true; } return false; // MP4 } private static void GetClue(string[,] map, MapSizeRecord mapSize, string[,] hiddenMap) // MP5 Task 2 { int rockRow = 0, rockColumn = 0, treasureRow = 0, treasureColumn = 0; int distanceFromRockNS = 0, distanceFromRockWE = 0; for (int row = 0; row < mapSize.rows; row++) { for (int column = 0; column < mapSize.columns; column++) // MP2 { if (map[row, column] == ROCK) // MP6 { rockRow = row; rockColumn = column; } } } for (int row = 0; row < mapSize.rows; row++) { for (int column = 0; column < mapSize.columns; column++) { if (hiddenMap[row, column] == TREASURE) // MP7 { treasureRow = row; treasureColumn = column; } } } distanceFromRockNS = treasureRow - rockRow; // MP3 if (distanceFromRockNS < 0) { distanceFromRockNS = -distanceFromRockNS; } distanceFromRockWE = treasureColumn - rockColumn; // MP8 if (distanceFromRockWE < 0) { distanceFromRockWE = -distanceFromRockWE; // MP10 } Console.WriteLine(\$"The treasure is {distanceFromRockNS} squares away from the rock in the North - South direction"); Console.WriteLine(\$"The treasure is {distanceFromRockWE} squares away from the rock in the East - West direction"); // MP9 } </pre>	12
----	---	---	----

```

static void PirateWalks(string[,] map, MapSizeRecord mapSize,
string[,] hiddenMap, ref PirateRecord pirate)
{
    bool obstacleInPath;
    bool validDistance;
    bool validDirection;
    string walkData;
    string direction = "";
    int row = 0;
    int column = 0;
    int numberOfSquares = 0;
    obstacleInPath = true;
    validDistance = false;
    validDirection = false;
    while (obstacleInPath || !validDistance || !validDirection)
    {
        Console.WriteLine("Enter length (1 to 9) and direction (N, NE,
E, SE, S, SW, W, NW): ");
        walkData = Console.ReadLine();
        row = pirate.row;
        column = pirate.column;
        (validDistance, numberOfSquares) =
CheckDistance(walkData[0].ToString());
        direction = walkData.Substring(1);
        (validDirection, row, column) = CheckDirection(direction,
row, column, numberOfSquares);
        if (row >= mapSize.rows || column >= mapSize.columns ||
row < 0 || column < 0)
        {
            validDirection = false;
            Console.WriteLine("Error");
        }
        if (validDirection)
        {
            obstacleInPath = CheckPath(map, pirate.row,
pirate.column, row, column, direction);
            if (obstacleInPath)
            {
                Console.WriteLine("Pirate can't walk this way as there
is an obstacle in the way");
            }
        }
    }
    Move(map, mapSize, ref pirate, row, column);
    if (WestOfHut(map, pirate)) // MP11 Task 3
    {
        GetClue(map, mapSize, hiddenMap); // MP12
    }
}

```

Java

03	1	<pre>String S = ""; while (!S.equals("x")) { // MP2 Console.write("Enter a word or phrase: "); S = Console.readLine(); // MP3 int Max = S.length() - 1; // MP4 boolean Matched = true; for (int i=0; i<Max+1; i++) { // MP5 char Letter1 = S.charAt(i); char Letter2 = S.charAt(Max - i); // MP1 if (Letter1 != Letter2) { // MP6 Matched = false; // MP7 } } if (Matched) { Console.println("Palindrome"); } else { Console.println("Not a palindrome"); // MP8 } }</pre>	8
----	---	---	---

15	1	<pre> void ResetPirateRecord(PirateRecord Pirate) { Pirate.Row = 0; Pirate.Column = 0; Pirate.Score = 100; Pirate.DigTime = 0.0f; Pirate.TreasureFound = false; Pirate.NumberOfCoinsFound = 0; Pirate.WalkTime = 0.0f; // MP1 Task 1 } void PirateWalks(String[][] Map, MapSizeRecord MapSize, String[][] HiddenMap, PirateRecord Pirate) { boolean ObstacleInPath = true; boolean ValidDistance = false; boolean ValidDirection = false; int Row = 0, Column = 0, NumberOfSquares = 0; String Direction = ""; while (ObstacleInPath !ValidDistance !ValidDirection) { Console.write("Enter length (1 to 9) and direction (N, NE, E, SE, S, SW, W, NW): "); String WalkData = Console.readLine(); Row = Pirate.Row; Column = Pirate.Column; MoveCheckRecord DistanceData = CheckDistance(String.valueOf(WalkData.charAt(0))); ValidDistance = DistanceData.valid; NumberOfSquares = DistanceData.numberOfSquares; Direction = WalkData.substring(1); MoveCheckRecord DirectionData = CheckDirection(Direction, Row, Column, NumberOfSquares); ValidDirection = DirectionData.valid; Row = DirectionData.row; Column = DirectionData.column; if (Row >= MapSize.Rows Column >= MapSize.Columns Row < 0 Column < 0) { ValidDirection = false; Console.WriteLine("Error"); } if (ValidDirection) { ObstacleInPath = CheckPath(Map, Pirate.Row, Pirate.Column, Row, Column, Direction); if (ObstacleInPath) { Console.WriteLine("Pirate can't walk this way as there is an obstacle in the way"); } } } Move(Map, MapSize, Pirate, Row, Column); if (Direction.length() == 1) { Pirate.WalkTime += NumberOfSquares; // MP2 Task 2 } else { Pirate.WalkTime += 1.4 * NumberOfSquares; // MP3 Task 2 } } void DisplayResults(PirateRecord Pirate) { if (Pirate.NumberOfCoinsFound > 0) { Console.WriteLine(Pirate.NumberOfCoinsFound + " gold coins found"); } Console.WriteLine(Pirate.DigTime + " hours spent digging"); } </pre>	
----	---	---	--

	<pre>Console.WriteLine(Pirate.WalkTime + " hours spent walking"); // MP4</pre> Task3 <pre>Console.WriteLine("The score is " + Pirate.Score); }</pre> Alternative answer 1 for mark points 2 and 3: <pre>if ("NSEW".contains(Direction)) { Pirate.WalkTime += NumberOfSquares; } else { Pirate.WalkTime += 1.4 * NumberOfSquares; }</pre> Alternative answer 2 for mark points 2 and 3: <pre>if (Direction == "N" Direction == "S" Direction == "E" Direction == "W") { Pirate.WalkTime += NumberOfSquares; } else { Pirate.WalkTime += 1.4 * NumberOfSquares; }</pre>	
--	--	--

16	1	<pre> void FindLandingPlace(String[][] Map, MapSizeRecord MapSize, PirateRecord Pirate) { boolean Found = false; int Row = 0; while (!Found && Row < (MapSize.Rows)) { int Column = 0; while (!Found && Column < (MapSize.Columns)) { if (Map[Row][Column].equals("X")) { Found = true; Pirate.Row = Row; Pirate.Column = Column; } Column += 1; } Row += 1; } DisplayMap(Map, MapSize); Console.WriteLine("X marks the spot where the pirate comes ashore"); Console.WriteLine(); Console.Write("Do you want the pirate to land elsewhere? (Y/N): "); String Answer = Console.ReadLine(); // MP3 if (Answer.equals("Y")) { Row = 0; int Column = 0; Console.Write("Which row is the pirate coming ashore? "); Row = Integer.parseInt(Console.ReadLine()); // MP1 Console.Write("Which column is the pirate coming ashore? "); Column = Integer.parseInt(Console.ReadLine()); // MP4 boolean OnSand = false; boolean NextToWater = false; if (Map[Row][Column].equals(SAND)) { // MP5 OnSand = true; } if (Map[Row-1][Column].equals(WATER) Map[Row+1][Column].equals(WATER)) { NextToWater = true; } if (Map[Row][Column].equals(WATER) Map[Row][Column-1].equals(WATER)) { NextToWater = true; // MP2 // MP6 } if (OnSand && NextToWater) { // MP7 Pirate.Row = Row; Pirate.Column = Column; // MP8 Map[Row][Column] = PIRATES; DisplayMap(Map, MapSize); // MP9 } else { Console.WriteLine("That was not a good landing place, so the pirate lands at X"); // MP10 } } } </pre>	10
----	---	--	----

17	1	<pre> boolean WestOfHut(String[][] Map, PirateRecord Pirate) { return Map[Pirate.Row][Pirate.Column + 1].equals(HUT); // MP1 MP4 Task 1 } void GetClue(String[][] Map, MapSizeRecord MapSize, String[][] HiddenMap, PirateRecord Pirate) { // MP5 Task 2 int RockRow = 0, RockColumn = 0, TreasureRow = 0, TreasureColumn = 0; for (int row = 0; row < MapSize.Rows; row++) { for (int column = 0; column < MapSize.Columns; column++) { // MP2 if (Map[row][column].equals(ROCK)) { // MP6 RockRow = row; RockColumn = column; } } } for (int row = 0; row < MapSize.Rows; row++) { for (int column=0; column < MapSize.Columns; column++) { if (HiddenMap[row][column].equals(TREASURE)) { // MP7 TreasureRow = row; TreasureColumn = column; } } } int DistanceFromRock_NS = TreasureRow - RockRow; // MP3 if (DistanceFromRock_NS < 0) { DistanceFromRock_NS *= -1; } int DistanceFromRock_WE = TreasureColumn - RockColumn; // MP6 if (DistanceFromRock_WE < 0) { DistanceFromRock_WE *= -1; // MP10 } Console.WriteLine("The treasure is " + DistanceFromRock_NS + " squares away from the rock in the North - South direction"); Console.WriteLine("The treasure is " + DistanceFromRock_WE + " squares away from the rock in the East - West direction"); // MP9 } void PirateWalks(String[][] Map, MapSizeRecord MapSize, String[][] HiddenMap, PirateRecord Pirate) { boolean ObstacleInPath = true; boolean ValidDistance = false; boolean ValidDirection = false; int Row = 0, Column = 0, NumberOfSquares = 0; String Direction = ""; while (ObstacleInPath !ValidDistance !ValidDirection) { Console.write("Enter length (1 to 9) and direction (N, NE, E, SE, S, SW, W, NW): "); String WalkData = Console.readLine(); Row = Pirate.Row; Column = Pirate.Column; MoveCheckRecord DistanceData = CheckDistance(String.valueOf(WalkData.charAt(0))); ValidDistance = DistanceData.valid; NumberOfSquares = DistanceData.numberOfSquares; Direction = WalkData.substring(1); MoveCheckRecord DirectionData = CheckDirection(Direction, Row, Column, NumberOfSquares); ValidDirection = DirectionData.valid; Row = DirectionData.row; Column = DirectionData.column; </pre>	12
----	---	--	----

	<pre> if (Row >= MapSize.Rows Column >= MapSize.Columns Row < 0 Column < 0) { ValidDirection = false; Console.WriteLine("Error"); } if (ValidDirection) { ObstacleInPath = CheckPath(Map, Pirate.Row, Pirate.Column, Row, Column, Direction); if (ObstacleInPath) { Console.WriteLine("Pirate can't walk this way as there is an obstacle in the way"); } } } Move(Map, MapSize, Pirate, Row, Column); if (WestOfHut(Map, Pirate)) { // MP11 Task 3 GetClue(Map, MapSize, HiddenMap, Pirate); // MP12 } } </pre>	
--	---	--