A-level
COMPUTER SCIENCE
7517/1

Paper 1

Mark scheme

June 2023

Version: 1.0 Final

236A7517/1/MS

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 2023

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 **06.1**, 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 AO1 (understanding)** 1. Check the queue is not already full; 2. Compare the value of the (rear) pointer with the maximum size of the array; 3. If equal then (rear) pointer becomes zero; **A.** index of the first position in the array instead of zero 4. Otherwise, add one to the (rear) pointer; 5. Insert new item in position indicated by (rear) pointer; **Alternative answer 1** 1. Check the queue is not already full; 2. Compare the value of the (rear) pointer with the maximum size of the array minus one; 3. If equal then (rear) pointer becomes one; **A.** index of the first position in the array instead of one 4. Otherwise, add one to the (rear) pointer; 5. Insert new item in position indicated by (rear) pointer; **Alternative answer 2** 1. Check the queue is not already full; 2. Add one to the (rear) pointer; 3. Compare the value of the (rear) pointer with the maximum size of the array; 4. If equal then (rear) pointer becomes zero; **A.** index of the first position in the array instead of zero 5. Insert new item in position indicated by (rear) pointer; **Alternative answer 3** 1. Check the queue is not already full; 2. Add one to the (rear) pointer; 3. Compare the value of the (rear) pointer with the maximum size of the array plus one; 4. If equal then (rear) pointer becomes one; **A.** index of the first position in the array instead of one 5. Insert new item in position indicated by (rear) pointer; **Alternative answer 4** 1. Check the queue is not already full; 2. Add one to the (rear) pointer; 3. Use modulus/modulo operator/function with new value of (rear) pointer; 4. Use modulus/modulo operator/function with maximum size of array; 5. Insert new item in position indicated by (rear) pointer; **Max 4** if any errors	**5**

Question			Marks
02	1	**Mark is for AO2 (analyse)** The colour is not yellow // the chosen shape was not the yellow circle // the colour is blue or pink;	1
02	2	**Mark is for AO2 (analyse)** The shape is not a square // the chosen shape was not the blue square // the shape is a triangle or circle;	1
02	3	**Mark is for AO2 (analyse)** Pink triangle;	1

<table>
<tr><th colspan="2">Question</th><th></th><th>Marks</th></tr>
<tr><td>03</td><td>1</td><td>All marks for AO1 (understanding)

A root (A. start) node; A. there is a parent-child relationship between nodes

Each node has no more than two child nodes; R. has two child nodes</td><td>2</td></tr>
<tr><td>03</td><td>2</td><td>All marks are for AO2 (apply)

<table>
<tr><td>1</td><td>0</td></tr>
<tr><td>2</td><td>-1</td></tr>
<tr><td>3</td><td>True</td></tr>
<tr><td>4</td><td>Current ← Tree[Current].Right</td></tr>
<tr><td>5</td><td>Current ← Tree[Current].Left</td></tr>
<tr><td>6</td><td>False</td></tr>
</table>

Note for examiners: answers are in pseudo-code so accept any reasonable representation (including use of string or integer values for rows 3 and 6).

Mark as follows:

1 mark: row one correct
1 mark: row two correct
1 mark: rows three and six correct
1 mark: rows four and five correct</td><td>4</td></tr>
<tr><td>03</td><td>3</td><td>Mark is for AO1 (knowledge)

$O(\log_2 n)$;

I. missing brackets
I. missing O
I. missing $_2$</td><td>1</td></tr>
<tr><td>03</td><td>4</td><td>Mark is for AO1 (understanding)

Every comparison halves the size of the binary tree to look at; (A. every comparison halves the size of the tree)</td><td>1</td></tr>
</table>

03	**5**	**Mark is for AO1 (understanding)** It does not have exponential (or worse) time complexity; It has a polynomial (or better) time complexity solution; **A.** It can be solved in a reasonable amount of time regardless of the problem/input size; **NE.** can be solved in a reasonable amount of time **Max 1**	**1**
03	**6**	**All marks AO1 (understanding)** Rules/knowledge (about the problem domain); Can be used to find a good/approximate but (probably) not optimal solution to a problem; Can reduce the size of the search/problem space // changing some constraints in the problem; **Max 2**	**2**
03	**7**	**All marks AO1 (knowledge)** As the size of the input/problem increases; the amount of time taken remains the same;	**2**

<table>
<tr><th colspan="2">Question</th><th></th><th>Marks</th></tr>
<tr><td>04</td><td>1</td><td>
All marks AO2 (analyse)
<table>
<tr><th>Language</th><th>Regular language (Y/N)?</th></tr>
<tr><td>Language A</td><td>N</td></tr>
<tr><td>Language B</td><td>Y</td></tr>
<tr><td>Language C</td><td>Y</td></tr>
<tr><td>Language D</td><td>Y</td></tr>
<tr><td>Language E</td><td>Y</td></tr>
<tr><td>Language F</td><td>Y</td></tr>
</table>
Mark as follows:

1 mark: any two rows correct
2 marks: any four rows correct
3 marks: all rows correct

A. any suitable alternative to N and Y
</td><td>3</td></tr>
<tr><td>04</td><td>2</td><td>
All marks AO2 (apply)

<code>a|ab|b+;;</code>

If final answer incorrect award a maximum of 1 mark for any of:
• <code>a|ab</code>
• <code>b+</code>

Alternative answer

<code>ab?|b+;;</code>

If final answer incorrect award a maximum of 1 mark for any of:
• <code>ab?</code>
• <code>b+</code>
</td><td>2</td></tr>
<tr><td>04</td><td>3</td><td>
Mark is for AO1 (knowledge)

The number of elements/items in a set; A. the size of a set
</td><td>1</td></tr>
</table>

<table>
<tr><th colspan="2">Question</th><th></th><th>Marks</th></tr>
<tr><td>05</td><td>1</td><td>
All marks AO2 (apply)
<table>
<tr><th colspan="10">Tape</th><th>Current State</th></tr>
<tr><td>…</td><td></td><td></td><td>#</td><td>0*</td><td>0</td><td>0</td><td>#</td><td></td><td>…</td><td>S0</td></tr>
<tr><td>…</td><td></td><td></td><td>#</td><td></td><td>0*</td><td>0</td><td>#</td><td></td><td>…</td><td>S2</td></tr>
<tr><td>…</td><td></td><td></td><td>#</td><td></td><td>0</td><td>0*</td><td>#</td><td></td><td>…</td><td>S1</td></tr>
<tr><td>…</td><td></td><td></td><td>#</td><td></td><td>0</td><td></td><td>#*</td><td></td><td>…</td><td>S2</td></tr>
<tr><td>…</td><td></td><td></td><td>#</td><td></td><td>0</td><td>*</td><td>#</td><td></td><td>…</td><td>S4</td></tr>
<tr><td>…</td><td></td><td></td><td>#</td><td></td><td>0*</td><td></td><td>#</td><td></td><td>…</td><td>S4</td></tr>
<tr><td>…</td><td></td><td></td><td>#</td><td>*</td><td>0</td><td></td><td>#</td><td></td><td>…</td><td>S4</td></tr>
<tr><td>…</td><td></td><td></td><td>#*</td><td></td><td>0</td><td></td><td>#</td><td></td><td>…</td><td>S4</td></tr>
<tr><td>…</td><td></td><td>*</td><td>#</td><td></td><td>0</td><td></td><td>#</td><td></td><td>…</td><td>S5</td></tr>
<tr><td>…</td><td></td><td>1</td><td>#*</td><td></td><td>0</td><td></td><td>#</td><td></td><td>…</td><td>S7</td></tr>
<tr><td>…</td><td></td><td>1</td><td>#</td><td>*</td><td>0</td><td></td><td>#</td><td></td><td>…</td><td>S0</td></tr>
<tr><td>…</td><td></td><td>1</td><td>#</td><td></td><td>0*</td><td></td><td>#</td><td></td><td>…</td><td>S0</td></tr>
<tr><td>…</td><td></td><td>1</td><td>#</td><td></td><td></td><td>*</td><td>#</td><td></td><td>…</td><td>S2</td></tr>
<tr><td>…</td><td></td><td>1</td><td>#</td><td></td><td></td><td></td><td>#*</td><td></td><td>…</td><td>S2</td></tr>
<tr><td>…</td><td></td><td>1</td><td>#</td><td></td><td></td><td>*</td><td>#</td><td></td><td>…</td><td>S4</td></tr>
<tr><td>…</td><td></td><td>1</td><td>#</td><td></td><td>*</td><td></td><td>#</td><td></td><td>…</td><td>S4</td></tr>
<tr><td>…</td><td></td><td>1</td><td>#</td><td>*</td><td></td><td></td><td>#</td><td></td><td>…</td><td>S4</td></tr>
<tr><td>…</td><td></td><td>1</td><td>#*</td><td></td><td></td><td></td><td>#</td><td></td><td>…</td><td>S4</td></tr>
<tr><td>…</td><td></td><td>1*</td><td>#</td><td></td><td></td><td></td><td>#</td><td></td><td>…</td><td>S5</td></tr>
<tr><td>…</td><td>*</td><td>1</td><td>#</td><td></td><td></td><td></td><td>#</td><td></td><td>…</td><td>S5</td></tr>
<tr><td>…</td><td>1</td><td>1*</td><td>#</td><td></td><td></td><td></td><td>#</td><td></td><td>…</td><td>S7</td></tr>
</table>
Mark as follows:

1 mark: tape and current state of first row correct.
1 mark: tape and current state of second and third rows correct.
1 mark: read/write head correct for first three rows.
1 mark: tape left of first hash symbol and current state for rows four to ten correct (denoted by blue outline in table above).
1 mark: tape and current state for last row correct.

A. any unambiguous way of denoting a blank cell on the tape
I. position of read/write head when awarding mark points 1, 2, 4 and 5

Max 4 if any errors
</td><td>5</td></tr>
<tr><td>05</td><td>2</td><td>
All marks AO2 (analyse)

When there are no zeros between the two hash symbols on the tape;
When there is a character other than zero (A. a one) between the two hash symbols on the tape;
If the machine was not in the start state (S0);

Max 2
</td><td>2</td></tr>
</table>

05	**3**	**All marks AO1 (knowledge)** A Turing machine that can execute/simulate the behaviour of any other Turing machine // can compute any computable sequence; Faithfully executes every single operation on the data precisely as the simulated TM would; (Note: must have idea of same process) Description of/Instructions for TM (and the TM's input) are stored on the (Universal Turing machine's) tape // The UTM acts as an interpreter; **A.** take any other TM and data as input **Max 2 marks** **Alternative definition:** A UTM, U, is an interpreter that reads the description of any arbitrary Turing machine M; and faithfully executes operations on data D precisely as M does.; The description is written at the beginning of the tape, followed by D.; **Max 2 marks**	**2**
05	**4**	**Mark is for AO1 (understanding)** It has an infinite amount of memory; **A.** the tape is infinitely long	**1**

Question			Marks
06	1	**4 marks for AO3 (design) and 8 marks for AO3 (programming)** **Mark Scheme**	12

Level	Description	Mark Range
4	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.	10–12
3	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 string, has at least one iterative structure and at least one selection structure and uses appropriate variables to store most of the needed data. An attempt has been made to test for most of the criteria for a valid string, although these may not work correctly under all circumstances. The solution demonstrates good design work as most of the correct design decisions have been made.	7–9
2	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.	4–6
1	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.	1–3

		Guidance **Evidence of AO3 design – 4 points:** Evidence of design to look for in responses: 1. Identifying that an iteration structure is needed that repeats a number of times based on the length of the string entered by the user. 2. Identifying that nested iteration is needed. 3. Identifying that an integer variable is needed to store the sum of the ASCII codes and that Boolean variable(s) (**A.** any suitable equivalent) are needed to track if there are duplicate characters and non-uppercase characters (**R.** if no attempt to use the Boolean variable (or equivalent) to indicate the result of at least one validation check). 4. Selection structure that checks if two characters in the string are the same **R.** if not inside their iteration structure (or equivalent) 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. User input being assigned to appropriate variable. 6. Correctly gets the ASCII code for a character. 7. Adds ASCII code for character to a total. 8. Correctly checks if every character is uppercase. **A.** checks every character is not lowercase 9. Correctly checks if a character is duplicated. **R.** if only checks if a character is a duplicate for some of the other characters in the string **R.** if will always say a character is a duplicate 10. Iteration structure that repeats until string is valid. **A.** if some validation checks are missing or incorrect **R.** if subsequent iterations would not work in same way e.g. because Boolean variables not reset inside iteration structure 11. Program rejects all strings that are less than five characters or more than seven characters in length. 12. Program works correctly under all circumstances. **Max 11** if any errors	

Question			Marks

06	2	**Mark is for AO3 (evaluate)**
		
		****** SCREEN CAPTURE ******
		*Must match code from **06.1**, including prompts on screen capture(s) matching those in code.*
		*Code for **06.1** must be sensible.*
		
		Screen captures showing the string(s) entered and result(s) of each of the tests;
		I. order of tests
		A. tests done individually or done as one extended test
		
		`Enter a string: BOIL`
		`not valid`
		`Enter a string: BRAisE`
		`not valid`
		`Enter a string: ROAST`
		`not valid`
		`Enter a string: BLANCH`
		`valid`
		
		`Enter a string: PRESSURECOOK`
		`not valid`
		`Enter a string:`
		
		Note for examiners: example screen captures shown here match the order of the test data given in the question but there is no requirement for the tests to be done in any particular order.
			1

<table>
<tr><th colspan="2">Question</th><th></th><th>Marks</th></tr>
<tr><td>07</td><td>1</td><td>All marks for AO2 (analyse)

<table>
<tr><th>Statement</th><th>True/False</th></tr>
<tr><td>Uses definite iteration</td><td>False</td></tr>
<tr><td>Uses nested iteration</td><td>True</td></tr>
<tr><td>Uses nested selection</td><td>False</td></tr>
<tr><td>Uses one or more global variables</td><td>False</td></tr>
<tr><td>Uses one or more local variables</td><td>True</td></tr>
<tr><td>Uses one or more named constants</td><td>False</td></tr>
</table>

A. any suitable alternative to True/False

Mark as follows:

1 mark: any four rows correct
2 marks: all rows correct</td><td>2</td></tr>
<tr><td>07</td><td>2</td><td>Mark is for AO2 (analyse)

13; A. –13</td><td>1</td></tr>
</table>

Question			Marks

08		**Marks are for AO1 (understanding)**
		
		Virtual methods can be overridden by the derived class //
		virtual methods do not have to be overridden //
		abstract methods must be overridden by the derived class;
		
		Virtual methods have an implementation/body //
		virtual methods contain code (with functionality) (in the base class) //
		abstract methods do not contain an implementation/body //
		abstract methods contain no code (with functionality) (in the base class) //
		abstract methods only contain a declaration (in the base class);
		
		Abstract methods can only be declared in abstract classes //
		virtual methods can be declared in abstract and non-abstract classes;
		
		Max 2
			2

Question			Marks
09	1	**Mark is for AO2 (analyse)** The implementation of the `Queue` data structure is not visible/known outside the class; Other parts of the program do not know that there is a data structure called `Queue`; Other parts of the program cannot access the data structure called `Queue`; **A.** the queue is private **R.** the queue is protected Other parts of the program do not know that a list data structure is used to store the move options; Other parts of the program do not know how the move options are stored; If the method used to represent the list was changed, the rest of the program would not need to be modified; **Max 1**	1
09	2	**Mark is for AO2 (analyse)** Can remove an item not at the front of the queue; Can use any of the (**A.** first three) items from the queue; **Max 1**	1
09	3	**Mark is for AO2 (analyse)** `Add`; **R.** if spelt incorrectly **R.** if any additional code **I.** case	1

Question			Marks
10	1	**Mark is for AO2 (analyse)** (runtime) error will occur if the following code is executed when square (being checked) does not contain a piece // it is necessary to check there is a piece in the square (before checking the type of the piece);	1
10	2	**Mark is for AO2 (apply)** `NOT Player1HasMirza OR NOT Player2HasMirza;` **R.** `Player1HasMirza = False OR Player2HasMirza = False`	1

Question			Marks

11	1	**Mark is for AO2 (analyse)**
		
		`DisplayBoard;`
		`DisplayFinalResult;`
		`DisplayState;`
		`GetPlayerStateAsString;`
		`GetQueueAsString;`
		`GetSquareReference;`
		`PlayGame;`
		
		Max 1
		
		R. if spelt incorrectly
		R. if any additional code
		I. case
			1

11	2	**Mark is for AO2 (analyse)** `Dastan;` **R.** if spelt incorrectly **R.** if any additional code **I.** case	1

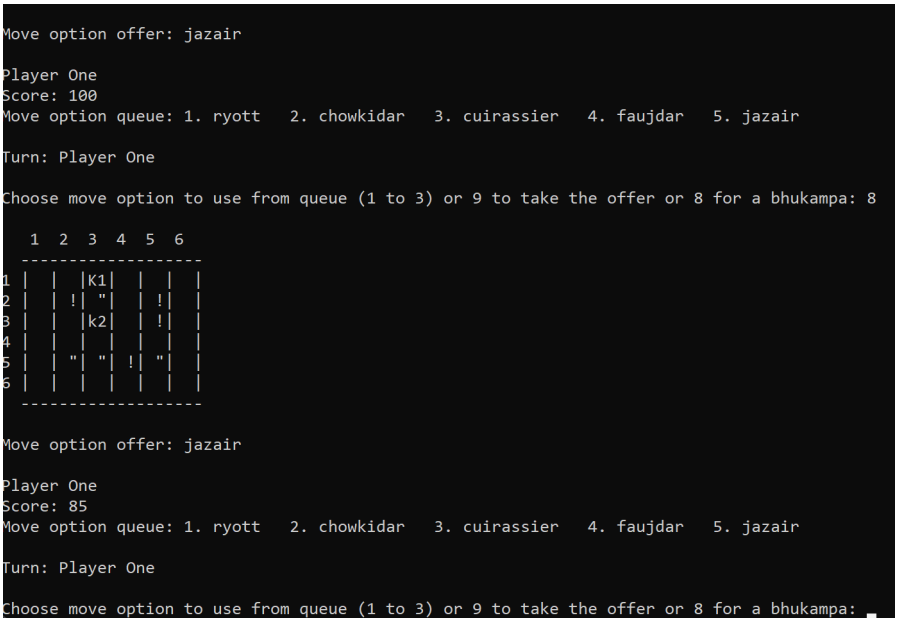
Question			Marks
12		**Mark is for AO2 (analyse)** So that player two's pieces move in the opposite direction to player one's; **A.** So that player two's pieces move up the board	1

Question			Marks
13	1	**All marks for AO3 (programming)** 1. Iterative structure contains code that gets the choice from the player; 2. One correct condition; 3. Both correct conditions and correct logic; 4. Displays error message under all correct circumstances and only under correct circumstances; **R.** message same as original prompt **Max 3** if code contains errors	4
13	2	**Mark is for AO3 (evaluate)** ****** SCREEN CAPTURE ****** *Must match code from **13.1**.* *Code for **13.1** must be sensible.* Screen capture showing message displayed when `6` is entered followed by `4` being entered and accepted;	1

```
   1  2  3  4  5  6
  ------------------
1 |  |  |K1|  |  |  |
2 |  | !| !| !| !|  |
3 |  |  |  |  |  |  |
4 |  |  |  |  |  |  |
5 |  | "| "| "| "|  |
6 |  |  |  |k2|  |  |
  ------------------

Move option offer: jazair

Player One
Score: 100
Move option queue: 1. ryott   2. chowkidar   3. cuirassier   4. faujdar   5. jazair

Turn: Player One

Choose move option to use from queue (1 to 3) or 9 to take the offer: 9
Choose the move option from your queue to replace (1 to 5): 6
Error - try again.
Choose the move option from your queue to replace (1 to 5): 4

   1  2  3  4  5  6
  ------------------
1 |  |  |K1|  |  |  |
2 |  | !| !| !| !|  |
3 |  |  |  |  |  |  |
4 |  |  |  |  |  |  |
5 |  | "| "| "| "|  |
6 |  |  |  |k2|  |  |
  ------------------

Move option offer: jazair

Player One
Score: 98
Move option queue: 1. ryott   2. chowkidar   3. cuirassier   4. jazair   5. jazair

Turn: Player One

Choose move option to use from queue (1 to 3) or 9 to take the offer:
```

Note for examiners: Bhukampa might be shown in list of choices or might not be.

Question			Marks
14	1	**All marks for AO3 (programming)** Mark points 1 to 6 refer to the new method `ProcessBhukampa`; mark points 7 to 9 refer to `PlayGame`. 1. Create a new method called `ProcessBhukampa`; **R.** other names for method; **I.** case and minor typos 2. Generates two random numbers; 3. Correct range for random numbers generated (0 to 35); **A.** 1 to 6 if generating random row/column position instead of position in `Board` if these are then used to create two valid square references and will be able to generate the full range of valid square references **A.** 1 to 6 if generating random row/column position instead of position in `Board` if these are then used to create an index between 0 and 35 4. Repeats until the two random numbers are different; 5. Swaps positions of two squares in `Board` list/array; **R.** only swapping the pieces that are in the two squares 6. Repeats attempt at (any of) their code for mark points 2 to 5 five times; 7. Call to new method from `PlayGame`; **R.** if not in iterative structure that gets move option from user 8. Selection structure in `PlayGame` with correct condition (= 8, or equivalent) 9. When bhukampa is chosen, player's score is decreased by 15 and call made to `DisplayState`; **R.** if (sometimes) executes when bhukampa not chosen **Max 8 marks** if code contains errors	9
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 option 8 being selected, player one score of 85 and new board state; **A.** score of 70 or 55 **Notes for examiners:** new board state is (partly) random so will not exactly match the one shown in this mark scheme.	1



Question			Marks
15	1	**All marks for AO3 (programming)** Mark points 1 to 6 relate to the `Gacaka` class. 1. Creating a new class called `Gacaka` that inherits from `Square`; **R.** other names for class; **I.** case and minor typos 2. Method called `SetPiece`/`GetPointsForOccupancy` created that overrides parent class method; 3. Message "`Trap!`" displayed in `SetPiece` method and piece is added to `Gacaka`; **R.** other messages **I.** case and minor typos **A.** added in appropriate place in `PlayGame` 4. Value of `PointsIfCaptured` for piece in `Gacaka` is increased by 2; **A.** making `PointsIfCaptured` public // new piece added to `Gacaka` which is same as original piece except `PointsIfCaptured` is two higher 5. Value of 0 returned by `GetPointsForOccupancy` if there is no piece in the gacaka; **R.** if always returns a value of 0 6. Value of –3 returned by `GetPointsForOccupancy` if there is a piece in the gacaka; **A.** if only returned for that player's turn or on both player's turn **R.** if always returns a value of -3 Mark points 7 to 8 relate to the `CreateBoard` method. 7. An object of type `Gacaka` is created; 8. The `Gacaka` object is added to the correct position in the `Board` list; **R.** if there are not exactly 36 objects in the `Board` list; **Max 7** if code contains errors (including not checking who the piece belongs to)	**8**

Question			Marks
15	2	**Mark is for AO3 (evaluate)** ****** SCREEN CAPTURE ****** *Must match code from **15.1**.* *Code for **15.1** must be sensible.* Screen capture(s) showing the correct final board state, the two player's scores and the message `Trap!` being displayed after the 2nd player's move; **A.** alternative messages if they match **15.1** **Note for examiners:** Bhukampa might be shown in list of choices or might not be.	1

```
Move option queue: 1. ryott   2. chowkidar   3. cuirassier   4. faujdar   5. jazair

Turn: Player One

Choose move option to use from queue (1 to 3) or 9 to take the offer or 8 for a bhukampa: 3
Enter the square containing the piece to move (row number followed by column number): 24
Enter the square to move to (row number followed by column number): 44
Trap!
New score: 95


   1  2  3  4  5  6
  ------------------
1 |  |  |K1|  |  |  |
2 |  | !| !|  | !|  |
3 |  |  |  |  |  |  |
4 |  |  |  | !|  |  |
5 |  | "| "| "| "|  |
6 |  |  |  |k2|  |  |
  ------------------

Move option offer: jazair

Player Two
Score: 100
Move option queue: 1. ryott   2. chowkidar   3. jazair   4. faujdar   5. cuirassier

Turn: Player Two

Choose move option to use from queue (1 to 3) or 9 to take the offer or 8 for a bhukampa: 1
Enter the square containing the piece to move (row number followed by column number): 54
Enter the square to move to (row number followed by column number): 44
Trap!
New score: 104


   1  2  3  4  5  6
  ------------------
1 |  |  |K1|  |  |  |
2 |  | !| !|  | !|  |
3 |  |  |  |  |  |  |
4 |  |  |  | "|  |  |
5 |  | "| "|  | "|  |
6 |  |  |  |k2|  |  |
  ------------------

Move option offer: jazair

Player One
Score: 95
Move option queue: 1. ryott   2. chowkidar   3. faujdar   4. jazair   5. cuirassier

Turn: Player One

Choose move option to use from queue (1 to 3) or 9 to take the offer or 8 for a bhukampa:
```

Question			Marks
16	1	**All marks for AO3 (programming)** Mark points 1 to 11 relate to the `GetNoOfPossibleMoves` method. 1. Creating a new method called `GetNoOfPossibleMoves` that takes `Board` / a list of squares as a parameter **R.** other method identifiers **I.** case and minor typos; 2. Iteration structure that repeats number of times based on size of board list; 3. Nested iteration structures that repeat correct number of times to check every combination of start and finish squares // nested iteration structures that repeat correct number of times to look at every combination of start square and legal move option. 4. Iteration structures that when combined will repeat enough times to check every combination of move option with the squares from their code for mark points 2 and 3; 5. Calculate the square reference for the start square; 6. Calculate the square reference for the finish square; 7. Calls the `CheckPlayerMove` / `CheckIfThereIsAMoveToSquare` method; **A.** suitable alternatives to calling method e.g. rewriting code from method 8. Checks if there is one of the player's pieces in the start square; 9. Checks if there is one of the opponent's pieces in the finish square and checks if the finish square does not contain a piece; 10. Adds one to the count of possible moves when (some) legal moves are found; **Note for examiners:** maximum of 1 mark for mark points 8 and 9 if the program would attempt to use a method/property for a piece in an empty square. `CheckSquareIsValid` completes all checks needed for mark points 8 and 9 (if called twice) but is not easily accessible from the `Player` class. **Note for examiners:** mark points 7 to 9 do not have to be inside iterative structures to be awarded 11. Call to `GetNoOfPossibleMoves` in appropriate place in `DisplayState` method; **A.** added to `DisplayBoard` instead of `DisplayState` 12. Value returned by `GetNoOfPossibleMoves` is displayed **Max 11** if code contains any errors	**12**

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.* Screen capture(s) showing that there are 52 legal moves for player two; `  1  2  3  4  5  6` `  ------------------` `1		|K1|		|` `2		!| !| !| !| |` `3						|` `4						|` `5		"| "| "| "| |` `6			|k2|	|` ` ------------------`  `Move option offer: jazair`  `Player One` `Score: 100` `Move option queue: 1. ryott 2. chowkidar 3. cuirassier 4. faujdar 5. jazair`  `Number of possible moves: 45` `Turn: Player One`  `Choose move option to use from queue (1 to 3) or 9 to take the offer or 8 for a bhukampa: 1` `Enter the square containing the piece to move (row number followed by column number): 22` `Enter the square to move to (row number followed by column number): 12` `New score: 104`  ` 1 2 3 4 5 6` ` ------------------` `1		!|K1|		|` `2			!| !| !| |` `3						|` `4						|` `5		"| "| "| "| |` `6			|k2|	|` ` ------------------`  `Move option offer: jazair`  `Player Two` `Score: 100` `Move option queue: 1. ryott 2. chowkidar 3. jazair 4. faujdar 5. cuirassier`  `Number of possible moves: 52` `Turn: Player Two`  `Choose move option to use from queue (1 to 3) or 9 to take the offer or 8 for a bhukampa:` **Note for examiners:** Bhukampa might be shown in list of choices or might not be.	1

VB.Net

Question			Marks

06	1	`unique = False`
		`s = "qqqqqqqq"`
		`valid = False`
		`total = 0`
		`While s.Length >= 8 or s.Length <= 4 Or Not unique Or Not valid Or total < 420 Or total > 600`
		`    total = 0`
		`    unique = True`
		`    valid = True`
		`    Console.Write("Enter a string: ")`
		`    s = Console.ReadLine()`
		`    For i = 0 To s.Length - 1`
		`        total += Asc(s(i))`
		`        If "ABCDEFGHIJKLMNOPQRSTUVWXYZ".Contains(s(i)) = False Then`
		`            valid = False`
		`        End If`
		`        For j = 0 To s.Length - 1`
		`            If i <> j And s(i) = s(j) Then`
		`                unique = False`
		`            End If`
		`        Next`
		`    Next`
		`    If s.Length >= 8 Or s.Length <= 4 Or Not unique Or Not valid Or total < 420 Or total > 600 Then`
		`        Console.WriteLine("not valid")`
		`    End If`
		`End While`
		`Console.WriteLine("valid")`
			12

13	1	`Private Sub UseMoveOptionOffer()`
		`  Dim ReplaceChoice As Integer = 0`
		`  **While ReplaceChoice < 1 Or ReplaceChoice > 5**`
		`    Console.Write("Choose the move option from your queue to replace (1 to 5): ")`
		`    ReplaceChoice = Console.ReadLine()`
		`    **If ReplaceChoice < 1 Or ReplaceChoice > 5 Then**`
		`      **Console.WriteLine("Error - try again.")**`
		`    **End If**`
		`  **End While**`
		`  CurrentPlayer.UpdateMoveOptionQueueWithOffer(ReplaceChoice - 1, CreateMoveOption(MoveOptionOffer(MoveOptionOfferPosition), CurrentPlayer.GetDirection()))`
		
		`    ...`
			4

14	1	`Public Sub PlayGame()`
		`  Dim GameOver As Boolean = False`
		`  While Not GameOver`
		`    DisplayState()`
		`    Dim SquareIsValid As Boolean = False`
			9

```
    Dim StartSquareReference, FinishSquareReference, Choice As
Integer
    Do
      Console.Write("Enter value between 1 and 3 to select move
option to use from queue or 9 to take the offer or 8 for a bhukampa:
")
      Choice = Console.ReadLine()
      If Choice = 9 Then
        UseMoveOptionOffer()
      End If
      If Choice = 8 Then
        ProcessBhukampa()
        CurrentPlayer.ChangeScore(-15)
        DisplayState()
      End If
    Loop Until Choice >= 1 And Choice <= 3
    While Not SquareIsValid
       ...

Private Sub ProcessBhukampa()
  Dim R As New Random
  Dim RNo1, RNo2 As Integer
  For Count = 1 To 5
    Do
      RNo1 = R.Next(0, 36)
      RNo2 = R.Next(0, 36)
    Loop Until RNo1 <> RNo2
    Dim Temp As Square = Board(RNo1)
    Board(RNo1) = Board(RNo2)
    Board(RNo2) = Temp
  Next
End Sub
```

15 **1**

```
Class Gacaka
  Inherits Square

  Public Overrides Sub SetPiece(ByVal P As Piece)
    MyBase.SetPiece(P)
    Console.WriteLine("Trap!")
    PieceInSquare = New Piece(P.GetTypeOfPiece(), P.GetBelongsTo(),
P.GetPointsIfCaptured() + 2, P.GetSymbol())
  End Sub

  Public Overrides Function GetPointsForOccupancy(ByVal
CurrentPlayer As Player) As Integer
    If PieceInSquare Is Nothing Then
      Return 0
    ElseIf CurrentPlayer.SameAs(PieceInSquare.BelongsTo()) Then
      Return -3
    Else
      Return 0
    End If
  End Function
End Class

Private Sub CreateBoard()
  Dim S As Square
  Board = New List(Of Square)
  For Row = 1 To NoOfRows
    For Column = 1 To NoOfColumns
```

8

```
      If Row = 1 And Column = NoOfColumns \ 2 Then
        S = New Kotla(Players(0), "K")
      ElseIf Row = NoOfRows And Column = NoOfColumns \ 2 + 1 Then
        S = New Kotla(Players(1), "k")
      ElseIf Row = 4 And Column = 4 Then
        S = New Gacaka()
      Else
        S = New Square()
      End If
      Board.Add(S)
    Next
  Next
End Sub
```

Alternative answer

```
Class Gacaka
  Inherits Square

  Public Overrides Sub SetPiece(P As Piece)
    PieceInSquare = P
    P.ChangePointsIfCaptured(2)
    Console.WriteLine("Trap!")
  End Sub

  Public Overrides Function GetPointsForOccupancy(CurrentPlayer As
Player) As Integer
    If PieceInSquare Is Nothing Then
      Return 0
    ElseIf CurrentPlayer.SameAs(PieceInSquare.BelongsTo()) Then
      Return -3
    Else
      Return 0
    End If
  End Function
End Class


Private Sub CreateBoard()
  Dim S As Square
  Board = New List(Of Square)
  For Row = 1 To NoOfRows
    For Column = 1 To NoOfColumns
      If Row = 1 And Column = NoOfColumns \ 2 Then
        S = New Kotla(Players(0), "K")
      ElseIf Row = NoOfRows And Column = NoOfColumns \ 2 + 1 Then
        S = New Kotla(Players(1), "k")
      ElseIf Row = 4 And Column = 4 Then
        S = New Gacaka()
      Else
        S = New Square()
      End If
      Board.Add(S)
    Next
  Next
End Sub


Class Piece
  ...
  Public Sub ChangePointsIfCaptured(ByVal Change As Integer)
    PointsIfCaptured += Change
```

		End Sub **...**	

16	**1**	**Public Function GetNoOfPossibleMoves(ByVal Board As List(Of Square)) As Integer**
		Dim NoOfPossibleMoves As Integer = 0
		For Count1 = 0 To 35
		If Board(Count1).GetPieceInSquare() IsNot Nothing Then
		If Board(Count1).GetPieceInSquare().GetBelongsTo().GetName() = Name Then
		For Count2 = 0 To 35
		Dim SSqRef As Integer = (Count1 \ 6 + 1) * 10 + Count1 Mod 6
		Dim FSqRef As Integer = (Count2 \ 6 + 1) * 10 + Count2 Mod 6
		For Count3 = 0 To 2
		If Queue.GetMoveOptionInPosition(Count3).CheckIfThereIsAMoveToSquare(SSqRef, FSqRef) Then
		If Board(Count2).GetPieceInSquare() Is Nothing Then
		NoOfPossibleMoves += 1
		ElseIf Board(Count2).GetPieceInSquare().GetBelongsTo().GetName() <> Name Then
		NoOfPossibleMoves += 1
		End If
		End If
		Next
		Next
		End If
		End If
		Next
		Return NoOfPossibleMoves
		End Function
		
		Private Sub DisplayState()
		DisplayBoard()
		Console.WriteLine("Move option offer: " & MoveOptionOffer(MoveOptionOfferPosition))
		Console.WriteLine()
		Console.WriteLine(CurrentPlayer.GetPlayerStateAsString())
		Console.WriteLine("Number of possible moves: " & CurrentPlayer.GetNoOfPossibleMoves(Board))
		Console.WriteLine("Turn: " & CurrentPlayer.GetName())
		Console.WriteLine()
		End Sub
			12

Python 3

Question			Marks

06	1	`unique = False`
		`s = "qqqqqqqq"`
		`valid = False`
		`total = 0`
		`while len(s) >= 8 or len(s) <= 4 or not unique or not valid or total < 420 or total > 600:`
		`    total = 0`
		`    unique = True`
		`    valid = True`
		`    s = input("Enter a string: ")`
		`    for i in range (len(s)):`
		`        total += ord(s[i])`
		`        if s[i] not in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":`
		`            valid = False`
		`        for j in range (len(s)):`
		`            if i != j and s[i] == s[j]:`
		`                unique = False`
		`    if len(s) >= 8 or len(s) <= 4 or not unique or not valid or total < 420 or total > 600:`
		`        print("not valid")`
		`print("valid")`
			12

13	1	`def __UseMoveOptionOffer(self):`
		`    ReplaceChoice  = 0`
		`    **while ReplaceChoice < 1 or ReplaceChoice > 5:**`
		`        ReplaceChoice = int(input("Choose the move option from your queue to replace (1 to 5): "))`
		`        **if ReplaceChoice < 1 or ReplaceChoice > 5:**`
		`            **print("Error - try again.")**`
		`    self._CurrentPlayer.UpdateMoveOptionQueueWithOffer(ReplaceChoice - 1,`
		`self.__CreateMoveOption(self._MoveOptionOffer[self._MoveOptionOfferPosition], self._CurrentPlayer.GetDirection()))`
		`    ...`
			4

14	1	`def PlayGame(self):`
		`    GameOver = False`
		`    while not GameOver:`
		`        self.__DisplayState()`
		`        SquareIsValid = False`
		`        Choice = 0`
		`        while Choice < 1 or Choice > 3:`
		`            Choice = int(input("Choose move option to use from queue (1 to 3) or 9 to take the offer **or 8 for a bhukampa**: "))`
		`            if Choice == 9:`
		`                self.__UseMoveOptionOffer()`
		`                self.__DisplayState()`
		` **if Choice == 8:**
			9

```
                self.ProcessBhukampa()
                self._CurrentPlayer.ChangeScore(-15)
                self.__DisplayState()
        while not SquareIsValid:
            StartSquareReference =
self.__GetSquareReference("containing the piece to move")
            SquareIsValid =
self.__CheckSquareIsValid(StartSquareReference, True)
        SquareIsValid = False
         ...

def ProcessBhukampa(self):
    for count in range (5):
        RNo1 = random.randint(0, 35)
        RNo2 = random.randint(0, 35)
        while RNo1 == RNo2:
            RNo1 = random.randint(0, 35)
            RNo2 = random.randint(0, 35)
        Temp = self._Board[RNo1]
        self._Board[RNo1] = self._Board[RNo2]
        self._Board[RNo2] = Temp
```

15 **1**

```
class Gacaka(Square):
    def SetPiece(self, P):
        super(Gacaka, self).SetPiece(P)
        print("Trap!")
        self._PieceInSquare = Piece(P.GetTypeOfPiece(),
P.GetBelongsTo(), P.GetPointsIfCaptured() + 2, P.GetSymbol())

    def GetPointsForOccupancy(self,CurrentPlayer):
        if self._PieceInSquare is None:
            return 0
        elif CurrentPlayer.SameAs(self._PieceInSquare.BelongsTo()):
            return -3
        else:
            return 0

    def __CreateBoard(self):
        for Row in range(1, self._NoOfRows + 1):
            for Column in range(1, self._NoOfColumns + 1):
                if Row == 1 and Column == self._NoOfColumns // 2:
                    S = Kotla(self._Players[0], "K")
                elif Row == self._NoOfRows and Column ==
self._NoOfColumns // 2 + 1:
                    S = Kotla(self._Players[1], "k")
                elif Row == 4 and Column == 4:
                    S = Gacaka()
                else:
                    S = Square()
                self._Board.append(S)
```

Alternative answer

8

```
class Gacaka(Square):
    def SetPiece(self, P):
        self._PieceInSquare = P
        P.ChangePointsIfCaptured(2)
        print("Trap!")

    def GetPointsForOccupancy(self,CurrentPlayer):
        if self._PieceInSquare is None:
            return 0
        elif CurrentPlayer.SameAs(self._PieceInSquare.BelongsTo()):
            return -3
        else:
            return 0

class Piece:
    ...
    def ChangePointsIfCaptured(self,Change):
        self._PointsIfCaptured += Change
```

16 **1**

```
def GetNoOfPossibleMoves(self, Board):
  NoOfPossibleMoves = 0
  for Count1 in range(36):
    if Board[Count1].GetPieceInSquare() is not None:
      if Board[Count1].GetPieceInSquare().GetBelongsTo().GetName()
== self.__Name:
        for Count2 in range(36):
          SSqRef = (Count1 // 6 + 1) * 10 + Count1 % 6
          FSqRef = (Count2 // 6 + 1) * 10 + Count2 % 6
          for Count3 in range(3):
            if
self.__Queue.GetMoveOptionInPosition(Count3).CheckIfThereIsAMoveToSq
uare(SSqRef, FSqRef):
                if Board[Count2].GetPieceInSquare() is None:
                  NoOfPossibleMoves += 1
                elif
Board[Count2].GetPieceInSquare().GetBelongsTo().GetName() !=
self.__Name:
                  NoOfPossibleMoves += 1
    return NoOfPossibleMoves


def __DisplayState(self):
  self.__DisplayBoard()
  print("Move option offer: " +
self._MoveOptionOffer[self._MoveOptionOfferPosition])
  print()
  print(self._CurrentPlayer.GetPlayerStateAsString())
  print("Number of possible moves: " +
str(self._CurrentPlayer.GetNoOfPossibleMoves(self._Board)))
  print("Turn: " + self._CurrentPlayer.GetName())
  print()
```

12

Python 2

Question			Marks

06	1	`unique = False`
		`s = "qqqqqqqq"`
		`valid = False`
		`total = 0`
		`while len(s) >= 8 or len(s) <= 4 or not unique or not valid or total < 420 or total > 600:`
		`    total = 0`
		`    unique = True`
		`    valid = True`
		`    s = raw_input("Enter a string: ")`
		`    for i in range (len(s)):`
		`        total += ord(s[i])`
		`        if s[i] not in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":`
		`            valid = False`
		`        for j in range (len(s)):`
		`            if i != j and s[i] == s[j]:`
		`                unique = False`
		`    if len(s) >= 8 or len(s) <= 4 or not unique or not valid or total < 420 or total > 600:`
		`        print "not valid"`
		`print "valid"`
			12

13	1	`def __UseMoveOptionOffer(self):`
		`    ReplaceChoice  = 0`
		`    **while ReplaceChoice < 1 or ReplaceChoice > 5:**`
		`        ReplaceChoice = int(raw_input("Choose the move option from your queue to replace (1 to 5): "))`
		`        **if ReplaceChoice < 1 or ReplaceChoice > 5:**`
		`            **print "Error - try again."**`
		`    self._CurrentPlayer.UpdateMoveOptionQueueWithOffer(ReplaceChoice - 1,`
		`self.__CreateMoveOption(self._MoveOptionOffer[self._MoveOptionOfferPosition], self._CurrentPlayer.GetDirection()))`
		`    ...`
			4

14	1	`def PlayGame(self):`
		`    GameOver = False`
		`    while not GameOver:`
		`        self.__DisplayState()`
		`        SquareIsValid = False`
		`        Choice = 0`
		`        while Choice < 1 or Choice > 3:`
		`            Choice = int(raw_input("Choose move option to use from queue (1 to 3) or 9 to take the offer **or 8 for a bhukampa**: "))`
		`            if Choice == 9:`
		`                self.__UseMoveOptionOffer()`
		`                self.__DisplayState()`
		` **if Choice == 8:**
			9

```
                self.ProcessBhukampa()
                self._CurrentPlayer.ChangeScore(-15)
                self.__DisplayState()
        while not SquareIsValid:
            StartSquareReference =
self.__GetSquareReference("containing the piece to move")
            SquareIsValid =
self.__CheckSquareIsValid(StartSquareReference, True)
        SquareIsValid = False
         ...

def ProcessBhukampa(self):
    for count in range (5):
        RNo1 = random.randint(0, 35)
        RNo2 = random.randint(0, 35)
        while RNo1 == RNo2:
            RNo1 = random.randint(0, 35)
            RNo2 = random.randint(0, 35)
        Temp = self._Board[RNo1]
        self._Board[RNo1] = self._Board[RNo2]
        self._Board[RNo2] = Temp
```

15 **1** **8**

```
class Gacaka(Square):
    def SetPiece(self, P):
        super(Gacaka, self).SetPiece(P)
        print "Trap!"
        self._PieceInSquare = Piece(P.GetTypeOfPiece(),
P.GetBelongsTo(), P.GetPointsIfCaptured() + 2, P.GetSymbol())

    def GetPointsForOccupancy(self, CurrentPlayer):
        if self._PieceInSquare is None:
            return 0
        elif CurrentPlayer.SameAs(self._PieceInSquare.BelongsTo()):
            return -3
        else:
            return 0

    def __CreateBoard(self):
        for Row in range(1, self._NoOfRows + 1):
            for Column in range(1, self._NoOfColumns + 1):
                if Row == 1 and Column == self._NoOfColumns // 2:
                    S = Kotla(self._Players[0], "K")
                elif Row == self._NoOfRows and Column ==
self._NoOfColumns // 2 + 1:
                    S = Kotla(self._Players[1], "k")
                elif Row == 4 and Column == 4:
                    S = Gacaka()
                else:
                    S = Square()
                self._Board.append(S)
```

Alternative answer

```
class Gacaka(Square):
    def SetPiece(self, P):
        self._PieceInSquare = P
        P.ChangePointsIfCaptured(2)
        print "Trap!"

    def GetPointsForOccupancy(self, CurrentPlayer):
        if self._PieceInSquare is None:
            return 0
        elif CurrentPlayer.SameAs(self._PieceInSquare.BelongsTo()):
            return -3
        else:
            return 0

class Piece(object):
    ...
    def ChangePointsIfCaptured(self, Change):
        self._PointsIfCaptured += Change
```

16 **1**

```
def GetNoOfPossibleMoves(self, Board):
  NoOfPossibleMoves = 0
  for Count1 in range(36):
    if Board[Count1].GetPieceInSquare() is not None:
      if Board[Count1].GetPieceInSquare().GetBelongsTo().GetName()
== self.__Name:
        for Count2 in range(36):
          SSqRef = (Count1 // 6 + 1) * 10 + Count1 % 6
          FSqRef = (Count2 // 6 + 1) * 10 + Count2 % 6
          for Count3 in range(3):
            if
self.__Queue.GetMoveOptionInPosition(Count3).CheckIfThereIsAMoveToSq
uare(SSqRef, FSqRef):
                if Board[Count2].GetPieceInSquare() is None:
                  NoOfPossibleMoves += 1
                elif
Board[Count2].GetPieceInSquare().GetBelongsTo().GetName() !=
self.__Name:
                  NoOfPossibleMoves += 1
    return NoOfPossibleMoves


def __DisplayState(self):
  self.__DisplayBoard()
  print "Move option offer: " +
self._MoveOptionOffer[self._MoveOptionOfferPosition]
  print
  print self._CurrentPlayer.GetPlayerStateAsString()
  print "Number of possible moves: " +
str(self._CurrentPlayer.GetNoOfPossibleMoves(self._Board))
  print "Turn: " + self._CurrentPlayer.GetName()
  print
```

12

C#

Question			Marks

06	1	`bool unique = false;`
		`string s = "qqqqqqqq";`
		`int total = 0;`
		`bool valid = true;`
		`while (s.Length >=8 || s.Length <=4 || !unique || !valid || total < 420 || total > 600)`
		`{`
		`  total = 0;`
		`  unique = true;`
		`  valid = true;`
		`  Console.Write("Enter a string: ");`
		`  s = Console.ReadLine();`
		`  for (int i = 0; i < s.Length; i++)`
		`  {`
		`  total += s[i];`
		`  if (!"ABCDEFGHIJKLMNOPQRSTUVWXYZ".Contains(s[i].ToString()))`
		`  valid = false;`
		`  for (int j = 0; j < s.Length; j++)`
		`  if (i != j && s[i] == s[j])`
		`  unique = false;`
		`  }`
		`  if (s.Length >= 8 || s.Length <= 4 || !unique || ! valid || total < 420 || total > 600)`
		`  Console.WriteLine("not valid");`
		`}`
		`Console.WriteLine("valid");`
		`Console.ReadLine();`
		`}`
			12

13	1	`private void UseMoveOptionOffer()`
		`{`
		`  int ReplaceChoice = 0;`
		`  while (ReplaceChoice < 1 || ReplaceChoice > 5)`
		`  {`
		`  Console.Write("Choose the move option from your queue to replace (1 to 5): ");`
		`  ReplaceChoice = Convert.ToInt32(Console.ReadLine());`
		`  if (ReplaceChoice < 1 || ReplaceChoice > 5)`
		`  {`
		`  Console.WriteLine("Error - try again.");`
		`  }`
		`  }`
		
		`CurrentPlayer.UpdateMoveOptionQueueWithOffer(ReplaceChoice - 1, CreateMoveOption(MoveOptionOffer[MoveOptionOfferPosition], CurrentPlayer.GetDirection()));`
		`  CurrentPlayer.ChangeScore(-(10 - (ReplaceChoice * 2)));`
		`  MoveOptionOfferPosition = RGen.Next(0, 5);`
			4

		} 	

14	**1**	public void PlayGame()
		{
		bool GameOver = false;
		while (!GameOver)
		{
		DisplayState();
		bool SquareIsValid = false;
		int Choice;
		do
		{
		Console.Write("Choose move option to use from queue (1 to 3) or 9 to take the offer **or 8 for a bhukampa:** ");
		Choice = Convert.ToInt32(Console.ReadLine());
		if (Choice == 9)
		{
		UseMoveOptionOffer();
		DisplayState();
		}
		if (Choice == 8)
		{
		ProcessBhukampa();
		CurrentPlayer.ChangeScore(-15);
		DisplayState();
		}
		}
		while (!(Choice >= 1 & Choice <= 3));
		int StartSquareReference = 0;
		while (!SquareIsValid)
		. . .
		
		private void ProcessBhukampa()
		{
		Random R = new Random();
		int RNo1, RNo2;
		for (var Count = 1; Count <= 5; Count++)
		{
		do
		{
		RNo1 = R.Next(0, 36);
		RNo2 = R.Next(0, 36);
		}
		while (RNo1 == RNo2);
		Square Temp = Board[RNo1];
		Board[RNo1] = Board[RNo2];
		Board[RNo2] = Temp;
		}
		}
			9

15	**1**	**class Gacaka : Square** **{**	**8**

```
    public override void SetPiece(Piece P)
    {
        base.SetPiece(P);
        Console.WriteLine("Trap!");
        PieceInSquare = new Piece(P.GetTypeOfPiece(),
P.GetBelongsTo(), P.GetPointsIfCaptured() + 2,
P.GetSymbol());
    }

    public override int GetPointsForOccupancy(Player
CurrentPlayer)
    {
        if (PieceInSquare == null)
        {
            return 0;
        }
        else if
(CurrentPlayer.SameAs(PieceInSquare.BelongsTo())
        {
            return -3;
        }
        else
        {
            return 0;
        }
    }
}

private void CreateBoard()
{
    Square S;
    Board = new List<Square>();
    for (var Row = 1; Row <= NoOfRows; Row++)
    {
        for (var Column = 1; Column <= NoOfColumns; Column++)
        {
            if (Row == 1 & Column == NoOfColumns / 2)
            {
                S = new Kotla(Players[0], "K");
            }
            else if (Row == NoOfRows & Column == NoOfColumns
/ 2 + 1)
            {
                S = new Kotla(Players[1], "k");
            }
            else if (Row == 4 && Column == 4)
            {
                S = new Gacaka();
            }
            else
            {
                S = new Square();
            }
```

```
            Board.Add(S);
        }
    }
}
```

Alternative answer

```
class Gacaka : Square
{
    public override void SetPiece(Piece P)
    {
        P.ChangePointsIfCaptured(2);
        Console.WriteLine("Trap!");
        PieceInSquare = P;
    }

    public override int GetPointsForOccupancy(Player
CurrentPlayer)
    {
        if (PieceInSquare == null)
        {
            return 0;
        }
        else if
(CurrentPlayer.SameAs(PieceInSquare.BelongsTo())
        {
            return -3;
        }
        else
        {
            return 0;
        }
    }
}
```

```
Class Piece
  ...
public void ChangePointsIfCaptured(int Change)
{
    PointsIfCaptured += Change;
}
  ...
```

16	**1**	`public int GetNoOfPossibleMoves(List<Square> Board)`
		`{`
		`  int NoOfPossibleMoves = 0;`
		`  for (int Count1 = 0; Count1 <= 35; Count1++)`
		`  {`
		`  if (Board[Count1].GetPieceInSquare() != null)`
		`  {`
			12

```
            if
(Board[Count1].GetPieceInSquare().GetBelongsTo().GetName() ==
Name)
            {
                for (int Count2 = 0; Count2 <= 35; Count2++)
                {
                    int SSqRef = (Count1 / 6 + 1) * 10 +
Count1 % 6;
                    int FSqRef = (Count2 / 6 + 1) * 10 +
Count2 % 6;
                    for (int Count3 = 0; Count3 <= 2;
Count3++)
                    {
                        if
(Queue.GetMoveOptionInPosition(Count3).CheckIfThereIsAMoveToS
quare(SSqRef, FSqRef))
                        {
                            if
(Board[Count2].GetPieceInSquare() == null)
                            {
                                NoOfPossibleMoves += 1;
                            }
                            else if
(Board[Count2].GetPieceInSquare().GetBelongsTo().GetName() !=
Name)
                            {
                                NoOfPossibleMoves += 1;
                            }
                        }
                    }
                }
            }
        }
    }
    return NoOfPossibleMoves;
}


private void DisplayState()
{
    DisplayBoard();
    Console.WriteLine("Move option offer: " +
MoveOptionOffer[MoveOptionOfferPosition]);
    Console.WriteLine();

Console.WriteLine(CurrentPlayer.GetPlayerStateAsString());
    Console.WriteLine("Number of possible moves: " +
CurrentPlayer.GetNoOfPossibleMoves(Board));
    Console.WriteLine("Turn: " + CurrentPlayer.GetName());
    Console.WriteLine();
}
```

Pascal/Delphi

Question			Marks
06	1		12

```
program Q06;

{$APPTYPE CONSOLE}

uses
  System.SysUtils;

var
  s: string;
  isValid: boolean;
  counts: array['A' .. 'Z'] of integer;
  ch: char;
  sum: integer;

begin
  repeat
    for ch := 'A' to 'Z' do
      counts[ch] := 0;
    write('Enter a string: ');
    readln(s);
    isValid := (s.length >= 5) and (s.Length <= 7);
    if isValid then
      begin
        for ch in s do
          if not (ch in ['A' .. 'Z']) then
            isValid := false;
        if isValid then
          begin
            for ch in s do
              begin
                inc(counts[ch], 1);
                if counts[ch] > 1 then
                  isValid := false
              end;
            if isValid then
              begin
                sum := 0;
                for ch in s do
                  inc(sum, ord(ch));
                isValid := (sum >= 420) and (sum <= 600)
              end
          end
      end;
    if isValid then
      writeln('Valid')
    else
      writeln('Invalid')
  until isValid;
  readln
```

		end.	

13	**1**	procedure Dastan.UseMoveOptionOffer();
		var
		  ReplaceChoice: integer;
		begin
		  ReplaceChoice := 0;
		  repeat
		  write('Choose the move option from your queue to replace (1 to 5): ');
		  readln(ReplaceChoice);
		  if (ReplaceChoice < 1) or (ReplaceChoice > 5) then
		  writeln('Error – try again.')
		  until (ReplaceChoice >= 1) and (ReplaceChoice <= 5);
		  CurrentPlayer.UpdateMoveOptionQueueWithOffer(ReplaceChoice - 1,
		CreateMoveOption(MoveOptionOffer[MoveOptionOfferPosition], CurrentPlayer.GetDirection()));
		  CurrentPlayer.ChangeScore(-(10 - (ReplaceChoice * 2)));
		  MoveOptionOfferPosition := random(5);
		end;
			4

14	**1**	**procedure Dastan.ProcessBhukampa();**
		
		var
		  count, square1, square2: integer;
		  tempSquare: Square;
		
		begin
		  for count := 1 to 5 do
		  begin
		  repeat
		  square1 := random(36);
		  square2 := random(36)
		  until square1 <> square2;
		  tempSquare := Board[square1];
		  Board[square1] := Board[square2];
		  Board[square2] := tempSquare
		  end
		end;
		
		procedure Dastan.PlayGame();
		var
		  GameOver: boolean;
		  SquareIsValid: boolean;
		  StartSquareReference, FinishSquareReference, Choice: integer;
		  MoveLegal: boolean;
		  PointsForPieceCapture: integer;
		begin
		  GameOver := false;
		  while not GameOver do
			9

```
  begin
    DisplayState();
    repeat
      write('Choose move option to use from queue (1 to 3) or
9 to take the offer or 8 for a bhukampa: ');
      readln(Choice);
      if Choice = 9 then
      begin
        UseMoveOptionOffer();
        DisplayState();
      end;
      if Choice = 8 then
      begin
        ProcessBhukampa();
        CurrentPlayer.ChangeScore(-15);
        DisplayState()
      end;
    until (Choice >= 1) and (Choice <= 3);
    SquareIsValid := false;
    while not SquareIsValid do
…
```

15 **1**

```
Gacaka = class(Square)
public
  procedure SetPiece(P: Piece); override;
  function GetPointsForOccupancy(CurrentPlayer: Player):
integer; override;
end;

procedure Gacaka.SetPiece(P: Piece);

begin
  inherited SetPiece(P);
  inc(P.PointsIfCaptured, 2);
  writeln('Trap!')
end;

function Gacaka.GetPointsForOccupancy(CurrentPlayer: Player):
integer;

begin
  if PieceInSquare = nil then
    result := 0
  else if CurrentPlayer.SameAs(PieceInSquare.GetBelongsTo())
then
    result := -3
  else
    result := 0
end;


procedure Dastan.CreateBoard();
var
```

8

```
  S: Square;
  Row, Column: integer;
begin
  Board := TList<Square>.Create();
  for Row := 1 to NoOfRows do
  begin
    for Column := 1 to NoOfColumns do
    begin
      if (Row = 1) and (Column = NoOfColumns div 2) then
      begin
        S := Kotla.Create(Players[0], 'K');
      end
      else if (Row = NoOfRows) and (Column = NoOfColumns div
2 + 1) then
      begin
        S := Kotla.Create(Players[1], 'k');
      end
      else if (Row = 4) and (Column = 4) then
        S := Gacaka.Create()
      else
      begin
        S := Square.Create();
      end;
      Board.Add(S);
    end;
  end;
end;
```

I. Constructor duplicating `Square.Create`

16 **1**

```
function Player.GetNoOfPossibleMoves(Board: TList<Square>):
integer;

var
  startIndex, endIndex: integer;
  startReference, endReference: integer;
  M: MoveOption;
  option: integer;

begin
  Result := 0;

  for startIndex := 0 to Board.Count - 1 do
  begin
    if Board[startIndex].GetPieceInSquare() <> nil then
      if
Board[startIndex].GetPieceInSquare().GetBelongsTo().SameAs(Se
lf) then
        for option := 0 to 2 do
        begin
          M := Queue.GetMoveOptionInPosition(option);
          for endIndex := 0 to Board.Count - 1 do
          begin
```

12

```
            startReference := (startIndex div 6 + 1) * 10 +
(startIndex mod 6 + 1);
            endReference := (endIndex div 6 + 1) * 10 +
(endIndex mod 6 + 1);
            if M.CheckIfThereIsAMoveToSquare(startReference,
endReference) then
            begin
              if Board[endIndex].GetPieceInSquare() = nil
then
                inc(Result, 1)
              else if not
Board[endIndex].GetPieceInSquare().GetBelongsTo().SameAs(Self
) then
                inc(Result, 1)
              end;
            end;
          end;
  end;
end;
```

Alternative answer

```
function Player.GetNoOfPossibleMoves(Board: TList<Square>):
integer;

var
  startIndex, endIndex: integer;
  M: MoveOption;
  option: integer;

  function indexToReference(index: integer): integer;

  begin
    Result := (index div 6 + 1) * 10 + (index mod 6 + 1)
  end;

begin
  Result := 0;

  for startIndex := 0 to Board.Count - 1 do
  begin
    if Board[startIndex].GetPieceInSquare() <> nil then
      if
Board[startIndex].GetPieceInSquare().GetBelongsTo().SameAs(Se
lf) then
        for option := 0 to 2 do
        begin
          M := Queue.GetMoveOptionInPosition(option);
          for endIndex := 0 to Board.Count - 1 do
```

```
            if
M.CheckIfThereIsAMoveToSquare(indexToReference(startIndex),
indexToReference(endIndex)) then
              begin
                if Board[endIndex].GetPieceInSquare() = nil
then
                  inc(Result, 1)
                else if not
Board[endIndex].GetPieceInSquare().GetBelongsTo().SameAs(Self
) then
                  inc(Result, 1)
              end;
        end;
  end;
end;

procedure Dastan.DisplayState();
begin
  DisplayBoard;
  writeln('Move option offer: ' +
MoveOptionOffer[MoveOptionOfferPosition]);
  writeln;
  writeln(CurrentPlayer.GetPlayerStateAsString());
  writeln('Number of possible moves: ' +
CurrentPlayer.GetNoOfPossibleMoves(Board).ToString());
  writeln('Turn: ', CurrentPlayer.GetName());
  writeln;
end;
```

Java

Question			Marks

06	1	`String input;`
		`boolean valid ;`
		`int codeTotal;`
		`do {`
		`  codeTotal = 0;`
		`  valid = true;`
		`  Console.writeLine("Enter string:");`
		`  input = Console.readLine();`
		`  if (input.length() < 5 || input.length() > 7) {`
		`  valid = false;`
		`  } else {`
		`  for (int i = 0; i < input.length(); i++) {`
		`  if (input.charAt(i) < 'A' || input.charAt(i) > 'Z') {`
		`  valid = false;`
		`  } else {`
		`  for (int j = 0; j < input.length(); j++) {`
		`  if (i != j && input.charAt(i) == input.charAt(j)) {`
		`  valid = false;`
		`  }`
		`  }`
		`  }`
		`  codeTotal += input.charAt(i);`
		`  }`
		`  if (codeTotal < 420 || codeTotal > 600) {`
		`  valid = false;`
		`  }`
		`  }`
		`  if (!valid) {`
		`  Console.writeLine("Not Valid");`
		`  }`
		`} while (!valid);`
		`Console.writeLine("Valid");`
			12

13	1	`private void useMoveOptionOffer() {`
		`  int replaceChoice;`
		`  **do {**`
		`  Console.write("Choose the move option from your queue to replace (1 to 5): ");`
		`  replaceChoice = Integer.parseInt(Console.readLine());`
		`  **if (replaceChoice < 1 || replaceChoice > 5) {**`
		`  **Console.writeLine("Error - try again.");**`
		`  **}**`
		`  **} while (replaceChoice < 1 || replaceChoice > 5);**`
		`  currentPlayer.updateMoveOptionQueueWithOffer(replaceChoice - 1, createMoveOption(moveOptionOffer.get(moveOptionOfferPosition), currentPlayer.getdirection()));`
		`  currentPlayer.changeScore(-(10 - (replaceChoice * 2)));`
			4

		`    moveOptionOfferPosition = rGen.nextInt(5);` `}`	

14	**1**	**`public void processBhukampa() {`**
		`    int squarePos1, squarePos2;`
		`    for (int count = 0; count < 5; count++) {`
		`        do {`
		`            squarePos1 = rGen.nextInt(board.size());`
		`            squarePos2 = rGen.nextInt(board.size());`
		`        } while (squarePos1 == squarePos2);`
		`        Square tempSquare = board.get(squarePos1);`
		`        board.set(squarePos1, board.get(squarePos2));`
		`        board.set(squarePos2, tempSquare);`
		`    }`
		`}`
		
		`public void playGame() {`
		`    boolean gameOver = false;`
		`    while (!gameOver) {`
		`        displayState();`
		`        boolean squareIsValid = false;`
		`        int choice;`
		`        do {`
		`            Console.write("Choose move option to use from queue (1 to 3) or 9 to take the offer` **`or 8 to use the Bhukampa:`** `");`
		`            choice = Integer.parseInt(Console.readLine());`
		`            if (choice == 9) {`
		`                useMoveOptionOffer();`
		`                displayState();`
		`            }`
		`            if (choice == 8) {`
		`                processBhukampa();`
		`                currentPlayer.changeScore(-15);`
		`                displayState();`
		`            }`
		`        } while (choice < 1 || choice > 3);`
			9

15	**1**	**`class Gacaka extends Square {`** **`    @Override`** **`    public void setPiece(Piece p){`** **`        super.setPiece(p);`** **`        Console.writeLine("Trap!");`** **`        p.pointsIfCaptured += 2;`** **`    }`** **`    @Override`** **`    public int getPointsForOccupancy(Player currentPlayer) {`** **`        if (pieceInSquare == null) {`** **`            return 0;`** **`        }`** **`        if (currentPlayer.sameAs(pieceInSquare.belongsTo())`** **`        {`** **`            return -3;`**	**8**

```
        }
        return 0;
    }
}
```

I. missing override annotation

```
private void createBoard() {
    Square s;
    board = new ArrayList<>();
    for (int row = 1; row <= noOfRows; row++) {
        for (int column = 1; column <= noOfColumns; column++) {
            if (row == 1 && column == noOfColumns / 2) {
                s = new Kotla(players.get(0), "K");
            } else if (row == noOfRows && column == noOfColumns / 2
+ 1) {
                s = new Kotla(players.get(1), "k");
            } else if (row == 4 && column == 4) {
                s = new Gacaka();
            } else {
                s = new Square();
            }
            board.add(s);
        }
    }
}
```

16 **1**

```
public int getNoOfPossibleMoves(List<Square> board) {
  int noOfMoves = 0;
  for (int startNo = 0; startNo < 36; startNo++) {
    if (board.get(startNo).getPieceInSquare() != null) {
      if (board.get(startNo).getPieceInSquare().getBelongsTo() ==
this) {
        for (int finishNo = 0; finishNo < 36; finishNo++) {
          for (int moveNo = 0; moveNo < 3; moveNo++) {
            int startRef = (startNo / 6 + 1) * 10 + startNo % 6;
            int finishRef = (finishNo / 6 + 1) * 10 + finishNo % 6;
            if
(queue.getMoveOptionInPosition(moveNo).checkIfThereIsAMoveToSquare(s
tartRef, finishRef)) {
              if (board.get(finishNo).getPieceInSquare() == null) {
                noOfMoves++;
              } else if
(board.get(finishNo).getPieceInSquare().getBelongsTo() != this) {
                noOfMoves++;
              }
            }
          }
        }
      }
    }
  }
  return noOfMoves;
}
```

12

```
private void displayState(){
  displayboard();
  Console.writeLine("Move option offer: " +
moveOptionOffer.get(moveOptionOfferPosition));
  Console.writeLine();
  Console.writeLine(currentPlayer.getPlayerStateAsString());
  Console.writeLine("Number of possible moves: " +
currentPlayer.getNoOfPossibleMoves(board));
  Console.writeLine("Turn: " + currentPlayer.getName());
  Console.writeLine();
}
```