AQA

AS
COMPUTER SCIENCE
7516/1

Paper 1

Mark scheme

June 2019

Version: 1.0 Final

I96A7516I/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 Assessment Writer.

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

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.

Page 5 – 18 contain the generic mark scheme.

Pages 19 to 39 contain the 'Program Source Codes' specific to the programming languages for questions 03.1, 14.1, 15.1, 16.1 and 17.2

pages 20 to 23 – VB.NET
pages 24 to 26 – PYTHON 2
pages 27 to 29 – PYTHON 3
pages 30 to 34 – PASCAL/Delphi
pages 35 to 39 – C#
pages 40 to 43 – JAVA

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 candidates' responses to the most appropriate level according to **its overall quality**, 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 **17.1**, the marks available for the AO3 elements are as follows:

AO3 (design) – 2 marks
AO3 (programming) – 7 marks
Where a candidate's answer only reflects one element of the AO, the maximum mark they can receive will be restricted accordingly.

Qu		Marks	

01	1	**All marks for AO1 (knowledge)**
		
		Difference:
		global variables accessible to all parts of the program
		// declared in main program block
		// local variables declared in subroutine
		// accessible only in the program block/subroutine in which it was declared;
		
		Reason:
		memory allocated to local variables can be reused when subroutine not in use;
		local variable only exists while the program block/subroutine is executing;
		using local variables makes subroutines self-contained;
		A prevents accidental changes;
		A easier debugging/maintenance;
		Max 2
			3

02	1	**All marks for AO2 (apply)**	**4**

x	**MyValue**	**y**	**y > -1 ? (True/False)**	**Numbers[y]**	**Numbers[y] < MyValue ? (True/False)**	**Numbers**		
						[0]	**[1]**	**[2]**
						43	17	85
1	17	0	True	43	False			
							(17)	
2	85	1	True	17	True			
								17
		0	True	43	True			
							43	
		-1	False					
						85		

1 mark for correct x column and `MyValue` column;
1 mark for correct y column (0, 1, 0, -1);
1 mark for correct Boolean values in columns 4 and 6;
A. TRUE/true, FALSE/false, Yes/No, Y/N and any other suitable indicators
1 mark for final contents of `Numbers` correct;

Qu		Marks	
02	2	**Mark is for AO2 (analyse)** sort from largest to smallest; **NE** Sort on its own **A** bubble sort;	**1**

03	1	**All marks for AO3 (programming)** **Mark as follows:** 1) Correct variable declarations for `NumberIn`, `NumberOut`, `Count`, `PartValue`; **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 prompt `"Enter a positive whole number: "` and `NumberIn` assigned value entered by user; 3) Correct initialisation of `NumberOut` and `Count`; 4) `WHILE` loop with syntax allowed by the programming language and correct condition for termination of the loop; 5) Correct incrementation of `Count` within `WHILE` loop; 6) Correct assignment to `PartValue` within `WHILE` loop but before `FOR` loop; 7) Correct updating of `NumberIn` within `WHILE` loop but before `FOR` loop; 8) `FOR` loop with syntax allowed by the programming language over correct range; 9) Correct assignment to `PartValue` inside `FOR` loop; 10) Correct calculation of `NumberOut` after `FOR` loop but within `WHILE` loop; 11) Output statement giving correct output after `WHILE` loop; **I.** Ignore minor differences in case and spelling **Max 10** if code does not function correctly	11

03	2	**Mark is for AO3 (evaluate)**
		
		****** SCREEN CAPTURE ******
		Must match code from 03.1, including prompts on screen capture matching those in code.
		Code for 03.1 must be sensible.
		
		Screen capture showing:
		'22' being entered and the message 'The result is: 10110' displayed
		'29' being entered and the message 'The result is: 11101' displayed
		'-1' being entered and the message 'The result is: 0' displayed
		
		`Enter a positive whole number: 22`
		`The result is:  10110`
		`>>>`
		`Enter a positive whole number: 29`
		`The result is:  11101`
		`>>>`
		`Enter a positive whole number: -1`
		`The result is:  0`
		`>>>`
			1

03	3	**Mark is for AO2 (analyse)** converts from (positive) decimal/denary to binary;	1

04	1	**Mark is for AO1 (understand)** `Valid` /`ValidPiece` /`ValidMove` /`Found` /`EndOfList` /`Jumping` /`GameEnd` /`FileFound`; **A** `CanJump`; **R.** if any additional code **R.** if spelt incorrectly **I.** case & spacing	1
04	2	**Mark is for AO1 (understand)** `ValidMove` /`ValidJump` /`ListEmpty`; **A.** `setUpBoard` (for Java only) **R.** if any additional code **R.** if spelt incorrectly **I.** case & spacing	1
05		**Mark is for AO1 (understand)** `MoveRecord` /`ListOfMoves`; **R.** if any additional code **R.** if spelt incorrectly **I.** case & spacing	1
06		**Mark is for AO1 (understand)** catch any file errors // stop program crashing if file doesn't exist;	1
07		**Mark is for AO2 (analyse)** positions of player A's pieces; **A** the contents of (the data structure/variable) `A` // pointer/address to `A` // `A`;	1

08		**All marks for AO1 (understand)** **Label** | **Description** (a) | no move possible (for player A) (b) | Player B moves (c) | Player B's turn (d) | no move possible (for player B) **1 mark** for 2 correct labels **2 marks** for 4 correct labels	2
09	1	**Mark is for AO2 (analyse)** `DisplayBoard;` **R.** if any additional code **R.** if spelt incorrectly **I.** case & spacing	1
09	2	**Mark is for AO2 (analyse)** `PrintResult;` **R.** if any additional code **R.** if spelt incorrectly **I.** case & spacing	1
09	3	**Mark is for AO2 (analyse)** `PrintLine;` **A.** `PrintRow` / `PrintMiddleRow`; **Max 1** **R.** if any additional code **R.** if spelt incorrectly **I.** case & spacing	1

10	1	**All marks for AO2 (analyse)** (row 0 column 0) is used to store the number of moves; (row 0 column 1) is used to store the number of pieces promoted to dames;	2
10	2	**Mark for AO2 (analyse)** There are (a maximum of) 12 pieces per player // each row stores data for each piece;	1
10	3	**All marks for AO2 (analyse)** rows 1 to 12 (in columns 0 and 1) store the coordinates/location of the pieces on the board; if coordinates are -1 then indicates no piece; (column 2) indicates if the piece is a dame // indicates state of each piece; **Max 2**	2
11		**1 mark is for AO1 (understand)** it checks whether the sum of row and column are an even number; **2 marks for AO2(analyse)** to blank out a square if it can't be used; to store a space if it can be used; **A** for 1 mark: creates the checker board pattern;	3
12		**All marks for AO2 (analyse)** it counts the number of moves that are possible at the current state of play; it acts as the index for the data structure `ListOfMoves`;	2
13		**All marks for AO2 (analyse)** 1) User is asked to enter a Piece ID; 2) the `ListOfMoves` is searched (sequentially) // linear search of `ListOfMoves` // `ListOfMoves` is stepped through; 3) for an occurrence of the piece ID entered; 4) until either the piece ID is found or the end of `ListOfMoves` is encountered; 5) if end of list is encountered user is asked again to enter the Piece ID;	5

14	**1**	**All marks for AO3 (programming)**	**3**
		Mark as follows:	
		1 mark for error codes 1 to 3 tested (using IF, nested IF or CASE)	
		A Error messages in a data structure and accessed via error code as index	
		1 mark for appropriate error messages **(A** similar wording but same meaning as)**:**	
		`'Error code 1 - Not a valid piece'`	
		`'Error code 2 - Not a valid move'`	
		`'Error code 3 - Not a number'`	
		1 mark outputting error code (1, 2, 3 or 4)	
		Note:	
		Messages such as "Error Code 1 – not valid" are not detailed enough and are not creditworthy.	
14	**2**	**Mark is for AO3 (evaluate)**	**1**
		****** SCREEN CAPTURE ******	
		Must match code from 14.1, including prompts on screen capture matching those in code.	
		Code for 14.1 must be sensible.	
		Screen capture showing:	
		`Next Player:  a`	
		`a5  can jump to  3  ,  2`	
		`a6  can jump to  3  ,  0`	
		`a6  can jump to  3  ,  4`	
		`a7  can jump to  3  ,  2`	
		`a7  can jump to  3  ,  6`	
		`a8  can jump to  3  ,  4`	
		`a9  can move to  3  ,  0`	
		`a9  can move to  3  ,  2`	
		`a10  can move to  3  ,  2`	
		`a10  can move to  3  ,  4`	
		`a11  can move to  3  ,  4`	
		`a11  can move to  3  ,  6`	
		`a12  can move to  3  ,  6`	
		`There are  13  possible moves`	
		`Which piece do you want to move? a4`	
		`Error code 1 - not a valid piece`	
		`Which piece do you want to move? a9`	
		`Which row do you want to move to? 3`	
		`Which column do you want to move to? 4`	
		`Error code 2 - not a valid move`	
		`Which row do you want to move to? a`	
		`Which column do you want to move to? 9`	
		`Error code 3 - not a number`	
		`Which row do you want to move to? 3`	
		`Which column do you want to move to? 0`	

15	**1**	**1 mark for AO3 (design) and 1 mark for AO3 (programming)** **Mark as follows:** **AO3 (design) – 1 mark:** 1) choosing the final if statement to amend; **AO3 (programming) – 1 mark:** 2) correct logic statement;	**2**
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:	**1**

```
Next Player:  a
a1  can move to  1  ,  0
a1  can move to  1  ,  2
a2  can move to  7  ,  0
a3  can move to  3  ,  6
a5  can move to  4  ,  3
a5  can jump to  5  ,  0
a6  can jump to  5  ,  2
a7  can move to  3  ,  4
a7  can move to  3  ,  6
There are  9  possible moves
Which piece do you want to move? a5
Which row do you want to move to? 5
Which column do you want to move to? 0
jumped over  b1

Player A:
[[9, 0, 0], [0, 1, 0], [6, 1, 0], [2, 7, 0], [0, 7, 0], [5,
0, 0], [3, 0, 0], [2, 5, 0], [1, 6, 0], [-1, -1, 0], [-1, -1,
0], [-1, -1, 0], [-1, -1, 0]]
Player B:
[[8, 0, 0], [4, 1, 0], [7, 2, 0], [5, 6, 0], [5, 4, 0], [1,
4, 0], [6, 3, 0], [6, 5, 0], [6, 7, 0], [-1, -1, 0], [-1, -1,
0], [-1, -1, 0], [-1, -1, 0]]

      0     1     2     3     4     5     6     7
   -------------------------------------------------
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
 0 |XXXXX|  a1 |XXXXX|     |XXXXX|     |XXXXX|  a4 |
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
   -------------------------------------------------
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
 1 |     |XXXXX|     |XXXXX|  b5 |XXXXX|  a8 |XXXXX|
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
   -------------------------------------------------
```

```
  |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
2 |XXXXX|     |XXXXX|     |XXXXX|  a7 |XXXXX|  a3 |
  |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
  -------------------------------------------------
  |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
3 |  a6 |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
  |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
  -------------------------------------------------
  |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
4 |XXXXX|  b1 |XXXXX|     |XXXXX|     |XXXXX|     |
  |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
  -------------------------------------------------
  |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
5 |  a5 |XXXXX|     |XXXXX|  b4 |XXXXX|  b3 |XXXXX|
  |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
  -------------------------------------------------
  |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
6 |XXXXX|  a2 |XXXXX|  b6 |XXXXX|  b7 |XXXXX|  b8 |
  |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
  -------------------------------------------------
  |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
7 |     |XXXXX|  b2 |XXXXX|     |XXXXX|     |XXXXX|
  |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
  -------------------------------------------------
```

16	**1**	**2 marks for AO3 (design) and 7 marks for AO3 (programming)**	**9**

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.	7–9
2	There is evidence that a line of reasoning has been partially followed. There is evidence of some appropriate design work.	4–6
1	An attempt has been made to write and amend the subroutine `PrintResult`. 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.	1–3

Marking guidance:

Evidence of AO3 design – 2 points:

Evidence of design to look for in response:

1) subroutine `CountNumberOfPieces` with interface so can be used for both A and B
2) A method for checking piece exists on board

Evidence of AO3 programming – 7 points:

Evidence of programming to look for in response:

3) in `CountNumberOfPieces` count variable initialised, updated and returned correctly
   A counting non-dames only
4) in `CountNumberOfPieces` loop through A/B/PlayersPieces
5) use value stored in A/B [0,1] as the number of dames
6) formula given in Q correctly programmed
7) comparing the two players' scores and output winner correctly
8) output calculated scores
9) sensible output in case of a draw

Note: output is the same whether or not Question 15 has been attempted.

16	**2**	**Mark is for AO3 (evaluate)** ****** SCREEN CAPTURE ****** Must match code from 16.1, including prompts on screen capture matching those	**1**

in code.
Code for 16.1 must be sensible.

Screen capture showing:

```
Enter the filename: game4.txt

Player A:
[[15, 2, 0], [1, 2, 0], [0, 3, 0], [0, 5, 0], [1, 6, 0], [0,
1, 1], [1, 0, 1], [1, 4, 0], [2, 7, 0], [2, 1, 0], [2, 3, 0],
[2, 5, 0], [3, 6, 0]]
Player B:
[[15, 0, 0], [4, 3, 0], [5, 0, 0], [5, 6, 0], [5, 4, 0], [4,
1, 0], [3, 2, 0], [6, 5, 0], [6, 7, 0], [3, 0, 0], [3, 4, 0],
[4, 5, 0], [4, 7, 0]]

      0     1     2     3     4     5     6     7
   -------------------------------------------------
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
 0 |XXXXX|  A5 |XXXXX|  a2 |XXXXX|  a3 |XXXXX|     |
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
   -------------------------------------------------
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
 1 |  A6 |XXXXX|  a1 |XXXXX|  a7 |XXXXX|  a4 |XXXXX|
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
   -------------------------------------------------
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
 2 |XXXXX|  a9 |XXXXX| a10 |XXXXX| a11 |XXXXX|  a8 |
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
   -------------------------------------------------
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
 3 |  b9 |XXXXX|  b6 |XXXXX| b10 |XXXXX| a12 |XXXXX|
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
   -------------------------------------------------
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
 4 |XXXXX|  b5 |XXXXX|  b1 |XXXXX| b11 |XXXXX| b12 |
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
   -------------------------------------------------
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
 5 |  b2 |XXXXX|     |XXXXX|  b4 |XXXXX|  b3 |XXXXX|
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
   -------------------------------------------------
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
 6 |XXXXX|     |XXXXX|     |XXXXX|  b7 |XXXXX|  b8 |
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
   -------------------------------------------------
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
 7 |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
   -------------------------------------------------
Next Player:  a
There are  0  possible moves
Game ended
A won this game with a score of  -17
B got a score of  3
```

17	1	**Mark is for AO2 (analyse)** `OpponentsPieces`; **R.** if any additional code **R.** if spelt incorrectly **I.** case & spacing	1
17	2	**2 marks for AO3 (design) and 7 marks for AO3 (programming)**	9

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.	7–9
2	There is evidence that a line of reasoning has been partially followed. There is evidence of some appropriate design work.	4–6
1	An attempt has been made to amend the subroutine `MoveDame`. 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.	1–3

Marking guidance:

Evidence of AO3 design – 2 points:

Evidence of design to look for in response:

1) validate that chosen piece is an opponent's existing piece
2) return updated `OpponentsPieces` from subroutine `MoveDame` (parameter by reference)

Evidence of AO3 programming – 7 points:

Evidence of programming to look for in response:

3) user prompt for which piece to take
4) extracting player letter from chosen piece
5) extracting index from chosen piece
6) retrieving coodinates from `OpponentsPieces`
7) set opponent's piece coordinates to -1
8) new dame's coordinates set to taken piece's coordinates
9) update parameters in calls to `MovePiece` in subroutine `MakeMove` (parameter by reference)

A. solutions that ask the user to input the row and column of the piece to be removed.

17	**3**	**Mark is for AO3 (evaluate)**	**1**

****** SCREEN CAPTURE ******
Must match code from 17.2, including prompts on screen capture matching those in code.
Code for 17.2 must be sensible.

Screen capture showing:

```
Do you want to load a saved game? (Y/N): y
Enter the filename: game3.txt

Player A:
[[8, 0, 0], [0, 1, 0], [6, 1, 0], [2, 7, 0], [0, 7, 0], [3, 2,
0], [3, 0, 0], [2, 5, 0], [1, 6, 0], [-1, -1, 0], [-1, -1, 0],
[-1, -1, 0], [-1, -1, 0]]
Player B:
[[8, 0, 0], [4, 1, 0], [7, 2, 0], [5, 6, 0], [5, 4, 0], [1, 4,
0], [6, 3, 0], [6, 5, 0], [6, 7, 0], [-1, -1, 0], [-1, -1, 0],
[-1, -1, 0], [-1, -1, 0]]

       0      1      2      3      4      5      6      7
    -----------------------------------------------------
    |XXXXX|      |XXXXX|      |XXXXX|      |XXXXX|      |
  0 |XXXXX|  a1  |XXXXX|      |XXXXX|      |XXXXX|  a4  |
    |XXXXX|      |XXXXX|      |XXXXX|      |XXXXX|      |
    -----------------------------------------------------
    |      |XXXXX|      |XXXXX|      |XXXXX|      |XXXXX|
  1 |      |XXXXX|      |XXXXX|  b5  |XXXXX|  a8  |XXXXX|
    |      |XXXXX|      |XXXXX|      |XXXXX|      |XXXXX|
    -----------------------------------------------------
    |XXXXX|      |XXXXX|      |XXXXX|      |XXXXX|      |
  2 |XXXXX|      |XXXXX|      |XXXXX|  a7  |XXXXX|  a3  |
    |XXXXX|      |XXXXX|      |XXXXX|      |XXXXX|      |
    -----------------------------------------------------
    |      |XXXXX|      |XXXXX|      |XXXXX|      |XXXXX|
  3 |  a6  |XXXXX|  a5  |XXXXX|      |XXXXX|      |XXXXX|
    |      |XXXXX|      |XXXXX|      |XXXXX|      |XXXXX|
    -----------------------------------------------------
    |XXXXX|      |XXXXX|      |XXXXX|      |XXXXX|      |
  4 |XXXXX|  b1  |XXXXX|      |XXXXX|      |XXXXX|      |
    |XXXXX|      |XXXXX|      |XXXXX|      |XXXXX|      |
    -----------------------------------------------------
    |      |XXXXX|      |XXXXX|      |XXXXX|      |XXXXX|
  5 |      |XXXXX|      |XXXXX|  b4  |XXXXX|  b3  |XXXXX|
    |      |XXXXX|      |XXXXX|      |XXXXX|      |XXXXX|
    -----------------------------------------------------
    |XXXXX|      |XXXXX|      |XXXXX|      |XXXXX|      |
  6 |XXXXX|  a2  |XXXXX|  b6  |XXXXX|  b7  |XXXXX|  b8  |
    |XXXXX|      |XXXXX|      |XXXXX|      |XXXXX|      |
    -----------------------------------------------------
    |      |XXXXX|      |XXXXX|      |XXXXX|      |XXXXX|
  7 |      |XXXXX|  b2  |XXXXX|      |XXXXX|      |XXXXX|
    |      |XXXXX|      |XXXXX|      |XXXXX|      |XXXXX|
    -----------------------------------------------------
Next Player:  a
a1  can move to  1  ,  0
```

```
a1  can move to  1  ,  2
a2  can move to  7  ,  0
a3  can move to  3  ,  6
a5  can move to  4  ,  3
a7  can move to  3  ,  4
a7  can move to  3  ,  6
a8  can jump to  3  ,  4
There are  8  possible moves
Which piece do you want to move? a2
Which row do you want to move to? 7
Which column do you want to move to? 0
Which piece do you want to take? b1

Player A:
[[9, 1, 0], [0, 1, 0], [4, 1, 1], [2, 7, 0], [0, 7, 0], [3, 2,
0], [3, 0, 0], [2, 5, 0], [1, 6, 0], [-1, -1, 0], [-1, -1, 0],
[-1, -1, 0], [-1, -1, 0]]
Player B:
[[8, 0, 0], [-1, -1, 0], [7, 2, 0], [5, 6, 0], [5, 4, 0], [1,
4, 0], [6, 3, 0], [6, 5, 0], [6, 7, 0], [-1, -1, 0], [-1, -1,
0], [-1, -1, 0], [-1, -1, 0]]

       0     1     2     3     4     5     6     7
    ------------------------------------------------
    |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
  0 |XXXXX| a1  |XXXXX|     |XXXXX|     |XXXXX| a4  |
    |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
    ------------------------------------------------
    |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
  1 |     |XXXXX|     |XXXXX| b5  |XXXXX| a8  |XXXXX|
    |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
    ------------------------------------------------
    |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
  2 |XXXXX|     |XXXXX|     |XXXXX| a7  |XXXXX| a3  |
    |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
    ------------------------------------------------
    |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
  3 | a6  |XXXXX| a5  |XXXXX|     |XXXXX|     |XXXXX|
    |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
    ------------------------------------------------
    |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
  4 |XXXXX| A2  |XXXXX|     |XXXXX|     |XXXXX|     |
    |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
    ------------------------------------------------
    |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
  5 |     |XXXXX|     |XXXXX| b4  |XXXXX| b3  |XXXXX|
    |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
    ------------------------------------------------
    |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
  6 |XXXXX|     |XXXXX| b6  |XXXXX| b7  |XXXXX| b8  |
    |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
    ------------------------------------------------
    |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
  7 |     |XXXXX| b2  |XXXXX|     |XXXXX|     |XXXXX|
    |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
    ------------------------------------------------
```

17	**4**	**Mark is for AO3 (evaluate)**	**1**

****** SCREEN CAPTURE ******
Must match code from 17.2, including prompts on screen capture matching those in code.
Code for 17.2 must be sensible.

Screen capture showing:

```
Next Player:  b
b2  can move to  6  ,  1
b3  can move to  4  ,  5
b3  can move to  4  ,  7
b4  can move to  4  ,  3
b4  can move to  4  ,  5
b5  can move to  0  ,  3
b5  can move to  0  ,  5
b6  can move to  5  ,  2
b6  can jump to  4  ,  5
b7  can jump to  4  ,  3
b7  can jump to  4  ,  7
b8  can jump to  4  ,  5
There are  12  possible moves
Which piece do you want to move? b5
Which row do you want to move to? 0
Which column do you want to move to? 3
Which piece do you want to take? a6

Player A:
[[9, 1, 0], [0, 1, 0], [4, 1, 1], [2, 7, 0], [0, 7, 0], [3, 2,
0], [-1, -1, 0], [2, 5, 0], [1, 6, 0], [-1, -1, 0], [-1, -1,
0], [-1, -1, 0], [-1, -1, 0]]
Player B:
[[9, 1, 0], [-1, -1, 0], [7, 2, 0], [5, 6, 0], [5, 4, 0], [3,
0, 1], [6, 3, 0], [6, 5, 0], [6, 7, 0], [-1, -1, 0], [-1, -1,
0], [-1, -1, 0], [-1, -1, 0]]

      0     1     2     3     4     5     6     7
   -------------------------------------------------
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
 0 |XXXXX|  a1 |XXXXX|     |XXXXX|     |XXXXX|  a4 |
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
   -------------------------------------------------
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
 1 |     |XXXXX|     |XXXXX|     |XXXXX|  a8 |XXXXX|
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
   -------------------------------------------------
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
 2 |XXXXX|     |XXXXX|     |XXXXX|  a7 |XXXXX|  a3 |
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
   -------------------------------------------------
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
 3 |  B5 |XXXXX|  a5 |XXXXX|     |XXXXX|     |XXXXX|
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
   -------------------------------------------------
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
 4 |XXXXX|  A2 |XXXXX|     |XXXXX|     |XXXXX|     |
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
   -------------------------------------------------
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
 5 |     |XXXXX|     |XXXXX|  b4 |XXXXX|  b3 |XXXXX|
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
   -------------------------------------------------
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
 6 |XXXXX|     |XXXXX|  b6 |XXXXX|  b7 |XXXXX|  b8 |
   |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|     |
   -------------------------------------------------
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
 7 |     |XXXXX|  b2 |XXXXX|     |XXXXX|     |XXXXX|
   |     |XXXXX|     |XXXXX|     |XXXXX|     |XXXXX|
   -------------------------------------------------
```

VB.Net

03 1 **11**

```
Dim NumberIn, NumberOut, Count, PartValue As Integer
Console.Write("Enter a positive whole number: ")
NumberIn = Console.ReadLine
NumberOut = 0
Count = 0
While NumberIn > 0
   Count += 1
   PartValue = NumberIn Mod 2
   NumberIn \= 2
   For i = 1 To Count - 1
       PartValue *= 10
   Next
   NumberOut += PartValue
End While
Console.WriteLine("The result is: " & NumberOut)
Console.ReadLine()
```

14 1 **3**

```
Sub DispayErrorCode(ByVal ErrorNumber As Integer)
    Console.WriteLine("Error Code " & ErrorNumber)
    If ErrorNumber = 1 Then
        Console.WriteLine("not a valid piece")
    ElseIf ErrorNumber = 2 Then
        Console.WriteLine("not a valid move")
    ElseIf ErrorNumber = 3 Then
        Console.WriteLine("not a number")
    ElseIf ErrorNumber = 4 Then
        Console.WriteLine("file error")
    End If
End Sub
```

15 1 **2**

```
Function ValidJump(ByVal Board(,) As String, ByVal
PlayersPieces(,) As Integer, ByVal Piece As String, ByVal
NewRow As Integer, ByVal NewColumn As Integer) As Boolean
    Dim Valid As Boolean
    Dim MiddlePiece, Player, OppositePiecePlayer,
MiddlePiecePlayer As String
    Dim Index, CurrentRow, CurrentColumn, MiddlePieceRow,
MiddlePieceColumn As Integer
    Valid = False
    MiddlePiece = ""
    Player = Left(Piece, 1).ToLower()
    If Len(Piece) = 2 Then
        Index = CInt(Right(Piece, 1))
    Else
        Index = CInt(Right(Piece, 2))
    End If
    If Player = "a" Then
        OppositePiecePlayer = "b"
    Else
        OppositePiecePlayer = "a"
    End If
    If NewRow >= 0 And NewRow < BoardSize And NewColumn
>= 0 And NewColumn < BoardSize Then
        If Board(NewRow, NewColumn) = Space Then
```

```
            CurrentRow = PlayersPieces(Index, Row)
            CurrentColumn = PlayersPieces(Index, Column)
            MiddlePieceRow = (CurrentRow + NewRow) \ 2
            MiddlePieceColumn = (CurrentColumn +
NewColumn) \ 2
            MiddlePiece = Board(MiddlePieceRow,
MiddlePieceColumn)
            MiddlePiecePlayer = Left(MiddlePiece,
1).ToLower()
            If MiddlePiecePlayer = OppositePiecePlayer
Then
                Valid = True
            End If
        End If
    End If
    Return Valid
End Function
```

Alternative logic statement:

```
MiddlePiecePlayer = OppositePiecePlayer and
MiddlePiecePlayer != ' ':
```

16	1	(code below)	**9**

```
Function CountNumberOfPieces(ByVal PlayersPieces(,) As
Integer) As Integer
    Dim Count As Integer = 0
    For Index = 1 To NumberOfPieces
        If PlayersPieces(Index, Row) > -1 Then
            Count += 1
        End If
    Next
    Return Count
End Function


Sub PrintResult(ByVal A(,) As Integer, ByVal B(,) As
Integer, ByVal NextPlayer As String)
    Console.WriteLine("Game ended")
    Dim TotalA As Integer = CountNumberOfPieces(A)
    Dim TotalB As Integer = CountNumberOfPieces(B)
    TotalA = A(0, 0) - TotalA - 10 * A(0, 1)
    TotalB = B(0, 0) - TotalB - 10 * B(0, 1)
    If TotalA < TotalB Then
        Console.WriteLine("A won this game with a score
of " & TotalA)
        Console.WriteLine("B got a score of " & TotalB)
    ElseIf TotalB < TotalA Then
        Console.WriteLine("B won this game with a score
of " & TotalB)
        Console.WriteLine("A got a score of ", TotalA)
    Else
        Console.WriteLine("It was a draw.  Both players
got a score of " & TotalA)
    End If
    PrintPlayerPieces(A, B)
End Sub
```

17	2		9

```
  Sub MoveDame(ByRef OpponentsPieces(,) As Integer, ByRef
NewRow As Integer, ByRef NewColumn As Integer, ByVal
Player As String)
    Dim Opponent As String = ""
    Dim ChosenPiece As String
    Dim Index As Integer
    NewRow = -1
    While Player = Opponent Or NewRow = -1
      Console.Write("Which piece do you want to take?")
      ChosenPiece = Console.ReadLine
      Opponent = ChosenPiece.Substring(0, 1).ToLower
      Index = CInt(ChosenPiece.Substring(1,
ChosenPiece.Length - 1))
      NewRow = OpponentsPieces(Index, Row)
      NewColumn = OpponentsPieces(Index, Column)
    End While
    OpponentsPieces(Index, Row) = -1
    OpponentsPieces(Index, Column) = -1
  End Sub

  Sub MakeMove(ByRef Board(,) As String, ByRef
PlayersPieces(,) As Integer, ByRef OpponentsPieces(,) As
Integer, ByVal ListOfMoves() As MoveRecord, ByVal
PieceIndex As Integer)
    Dim Piece, MiddlePiece As String
    Dim NewRow, NewColumn, PlayersPieceIndex, CurrentRow,
CurrentColumn, MiddlePieceRow, MiddlePieceColumn As
Integer
    Dim Jumping As Boolean
    PlayersPieces(0, 0) = PlayersPieces(0, 0) + 1
    If PieceIndex > 0 Then
      Piece = ListOfMoves(PieceIndex).Piece
      NewRow = ListOfMoves(PieceIndex).NewRow
      NewColumn = ListOfMoves(PieceIndex).NewColumn
      If Len(Piece) = 2 Then
        PlayersPieceIndex = CInt(Right(Piece, 1))
      Else
        PlayersPieceIndex = CInt(Right(Piece, 2))
      End If
      CurrentRow = PlayersPieces(PlayersPieceIndex, Row)
      CurrentColumn = PlayersPieces(PlayersPieceIndex,
Column)
      Jumping = ListOfMoves(PieceIndex).CanJump
      MovePiece(Board, PlayersPieces, OpponentsPieces,
Piece, NewRow, NewColumn)
      If Jumping Then
        MiddlePieceRow = (CurrentRow + NewRow) \ 2
        MiddlePieceColumn = (CurrentColumn + NewColumn) \
2
        MiddlePiece = Board(MiddlePieceRow,
MiddlePieceColumn)
        Console.WriteLine("jumped over " & MiddlePiece)
      End If
    End If
  End Sub
  Sub MovePiece(ByRef Board(,) As String, ByRef
PlayersPieces(,) As Integer, ByRef OpponentsPieces(,) As
Integer, ByVal ChosenPiece As String, ByVal NewRow As
Integer, ByVal NewColumn As Integer)
    Dim Index, CurrentRow, CurrentColumn As Integer
    Dim Player As String
    If Len(ChosenPiece) = 2 Then
      Index = CInt(Right(ChosenPiece, 1))
    Else
      Index = CInt(Right(ChosenPiece, 2))
    End If
    CurrentRow = PlayersPieces(Index, Row)
    CurrentColumn = PlayersPieces(Index, Column)
    Board(CurrentRow, CurrentColumn) = Space
    If NewRow = BoardSize - 1 And PlayersPieces(Index,
Dame) = 0 Then
      Player = "a"
      PlayersPieces(0, 1) = PlayersPieces(0, 1) + 1
      PlayersPieces(Index, Dame) = 1
      ChosenPiece = ChosenPiece.ToUpper()
      MoveDame(OpponentsPieces, NewRow, NewColumn,
Player)
    Else
      If NewRow = 0 And PlayersPieces(Index, Dame) = 0
Then
        Player = "b"
        PlayersPieces(0, 1) = PlayersPieces(0, 1) + 1
        PlayersPieces(Index, Dame) = 1
        ChosenPiece = ChosenPiece.ToUpper()
        MoveDame(OpponentsPieces, NewRow, NewColumn,
Player)
      End If
    End If
    PlayersPieces(Index, Row) = NewRow
    PlayersPieces(Index, Column) = NewColumn
    Board(NewRow, NewColumn) = ChosenPiece
  End Sub
```

Python 3

03	1	`NumberIn = int(input('Enter a positive whole number: '))`
		`NumberOut = 0`
		`Count = 0`
		`while NumberIn > 0:`
		`  Count += 1`
		`  PartValue = NumberIn % 2`
		`  NumberIn = NumberIn // 2`
		`  for i in range(1, Count):`
		`    PartValue = PartValue * 10`
		`  NumberOut = NumberOut + PartValue`
		`print('The result is: ', NumberOut)`
			11

14	1	`def DisplayErrorCode(ErrorNumber):`
		`  print('Error Code ', ErrorNumber, ' - ', end='')`
		`  if ErrorNumber == 1:`
		`    print('not a valid piece')`
		`  elif ErrorNumber == 2:`
		`    print('not a valid move')`
		`  elif ErrorNumber == 3:`
		`    print('not a number')`
		`  elif ErrorNumber == 4:`
		`    print('file error')`
			3

15	1	`def ValidJump(Board, PlayersPieces, Piece, NewRow,`
		`NewColumn):`
		`  Valid = False`
		`  MiddlePiece = ''`
		`  Player = Piece[0].lower()`
		`  Index = int(Piece[1:])`
		`  if Player == 'a':`
		`    OppositePiecePlayer = 'b'`
		`  else:`
		`    OppositePiecePlayer = 'a'`
		`  if NewRow in range(BOARD_SIZE) and NewColumn in`
		`range(BOARD_SIZE):`
		`    if Board[NewRow][NewColumn] == SPACE:`
		`      CurrentRow = PlayersPieces[Index][ROW]`
		`      CurrentColumn = PlayersPieces[Index][COLUMN]`
		`      MiddlePieceRow = (CurrentRow + NewRow) // 2`
		`      MiddlePieceColumn = (CurrentColumn + NewColumn) // 2`
		`      MiddlePiece =`
		`Board[MiddlePieceRow][MiddlePieceColumn]`
		`      MiddlePiecePlayer = MiddlePiece[0].lower()`
		`      if ` **`MiddlePiecePlayer == OppositePiecePlayer`**`:`
		`        Valid = True`
		`  return Valid`
		
		Alternative logic statement:
		
		`MiddlePiecePlayer == OppositePiecePlayer and`
		`MiddlePiecePlayer != ' ':`
			2

16	1	**`def CountNumberOfPieces(PlayersPieces):`**
		`  Count = 0`
		`  for Index in range(1, NUMBER_OF_PIECES + 1):`
		`    if PlayersPieces[Index][ROW] > -1:`
			9

```
# allow COLUMN instead of ROW
      Count += 1
  return Count

def PrintResult(A, B, NextPlayer):
  print('Game ended')
  TotalA = CountNumberOfPieces(A)
  TotalB = CountNumberOfPieces(B)
  TotalA = A[0][0] - TotalA - 10 * A[0][1]
  TotalB = B[0][0] - TotalB - 10 * B[0][1]
  if TotalA < TotalB:
    print('A won this game with a score of ', TotalA)
    print('B got a score of ', TotalB)
  elif TotalB < TotalA:
    print('B won this game with a score of ', TotalB)
    print('A got a score of ', TotalA)
  else:
    print('it was a draw. Both players got a score of ',
TotalA)
  PrintPlayerPieces(A, B)
```

17	2	(code below)	9

```
def MoveDame(Player, OpponentsPieces):
  NewRow = -1
  Opponent = ''
  while Player == Opponent or NewRow == -1:
    ChosenPiece = input('Which piece do you want to take?
')
    Opponent = ChosenPiece[0].lower()
    Index = int(ChosenPiece[1:])
    NewRow = OpponentsPieces[Index][ROW]
    NewColumn = OpponentsPieces[Index][COLUMN]
  OpponentsPieces[Index][ROW] = -1
  OpponentsPieces[Index][COLUMN] = -1
  return NewRow, NewColumn, OpponentsPieces

def MovePiece(Board, PlayersPieces, OpponentsPieces,
ChosenPiece, NewRow, NewColumn):
  Index = int(ChosenPiece[1:])
  CurrentRow = PlayersPieces[Index][ROW]
  CurrentColumn = PlayersPieces[Index][COLUMN]
  Board[CurrentRow][CurrentColumn] = SPACE

  if NewRow == BOARD_SIZE - 1 and
PlayersPieces[Index][DAME] == 0:
    Player = 'a'
    PlayersPieces[0][1] += 1
    PlayersPieces[Index][DAME] = 1
    ChosenPiece = ChosenPiece.upper()
    NewRow, NewColumn, OpponentsPieces = MoveDame(Player,
OpponentsPieces)
  elif NewRow == 0 and PlayersPieces[Index][DAME] == 0:
    Player = 'b'
    PlayersPieces[0][1] += 1
    PlayersPieces[Index][DAME] = 1
    ChosenPiece = ChosenPiece.upper()
    NewRow, NewColumn, OpponentsPieces = MoveDame(Player,
OpponentsPieces)
```

```
  PlayersPieces[Index][ROW] = NewRow
  PlayersPieces[Index][COLUMN] = NewColumn
  Board[NewRow][NewColumn] = ChosenPiece
  return Board, PlayersPieces, OpponentsPieces

def MakeMove(Board, PlayersPieces, OpponentsPieces,
ListOfMoves, PieceIndex):
  PlayersPieces[0][0] += 1
  if PieceIndex > 0:
    Piece = ListOfMoves[PieceIndex].Piece
    NewRow = ListOfMoves[PieceIndex].NewRow
    NewColumn = ListOfMoves[PieceIndex].NewColumn
    PlayersPieceIndex = int(Piece[1:])
    CurrentRow = PlayersPieces[PlayersPieceIndex][ROW]
    CurrentColumn =
PlayersPieces[PlayersPieceIndex][COLUMN]
    Jumping = ListOfMoves[PieceIndex].CanJump
    Board, PlayersPieces, OpponentsPieces =
MovePiece(Board, PlayersPieces, OpponentsPieces, Piece,
NewRow, NewColumn)
    if Jumping:
      MiddlePieceRow = (CurrentRow + NewRow) // 2
      MiddlePieceColumn = (CurrentColumn + NewColumn) // 2
      MiddlePiece =
Board[MiddlePieceRow][MiddlePieceColumn]
      print('jumped over ', MiddlePiece)
  return Board, PlayersPieces, OpponentsPieces
```

Python 2

03 1

```
NumberIn = int(raw_input('Enter a positive whole number: 
'))
NumberOut = 0
Count = 0
while NumberIn > 0:
  Count += 1
  PartValue = NumberIn % 2
  NumberIn = NumberIn // 2
  for i in range(1, Count):
    PartValue = PartValue * 10
  NumberOut = NumberOut + PartValue
print 'The result is: ', NumberOut
```

11

14 1

```
def DisplayErrorCode(ErrorNumber):
  print 'Error Code ', ErrorNumber
  if ErrorNumber == 1:
    print 'not a valid piece'
  elif ErrorNumber == 2:
    print 'not a valid move'
  elif ErrorNumber == 3:
    print 'not a number'
  elif ErrorNumber == 4:
    print 'file error'
```

3

15 1

```
def ValidJump(Board, PlayersPieces, Piece, NewRow, 
NewColumn):
  Valid = False
  MiddlePiece = ''
  Player = Piece[0].lower()
  Index = int(Piece[1:])
  if Player == 'a':
    OppositePiecePlayer = 'b'
  else:
    OppositePiecePlayer = 'a'
  if NewRow in range(BOARD_SIZE) and NewColumn in 
range(BOARD_SIZE):
    if Board[NewRow][NewColumn] == SPACE:
      CurrentRow = PlayersPieces[Index][ROW]
      CurrentColumn = PlayersPieces[Index][COLUMN]
      MiddlePieceRow = (CurrentRow + NewRow) // 2
      MiddlePieceColumn = (CurrentColumn + NewColumn) // 2
      MiddlePiece = 
Board[MiddlePieceRow][MiddlePieceColumn]
      MiddlePiecePlayer = MiddlePiece[0].lower()
      if MiddlePiecePlayer == OppositePiecePlayer and 
MiddlePiecePlayer != ' ':
        Valid = True
  return Valid
```

2

16 1

```
def CountNumberOfPieces(PlayersPieces):
  Count = 0
  for Index in range(1, NUMBER_OF_PIECES + 1):
    if PlayersPieces[Index][ROW] > -1:
# allow COLUMN instead of ROW
      Count += 1
  return Count
```

9

```
def PrintResult(A, B, NextPlayer):
  print 'Game ended'
  TotalA = CountNumberOfPieces(A)
  TotalB = CountNumberOfPieces(B)
  TotalA = A[0][0] - TotalA - 10 * A[0][1]
  TotalB = B[0][0] - TotalB - 10 * B[0][1]
  if TotalA < TotalB:
    print 'A won this game with a score of ', TotalA
    print 'B got a score of ', TotalB
  elif TotalB < TotalA:
    print 'B won this game with a score of ', TotalB
    print 'A got a score of ', TotalB
  else:
    print 'it was a draw. Both players got a score of ',
TotalA
  PrintPlayerPieces(A, B)
```

17 | 2 | 9

```
def MoveDame(Player, OpponentsPieces):
  NewRow = -1
  Opponent = ''
  while Player == Opponent or NewRow == -1:
    ChosenPiece = raw_input('Which piece do you want to
take? ')
    Opponent = ChosenPiece[0].lower()
    Index = int(ChosenPiece[1:])
    NewRow = OpponentsPieces[Index][ROW]
    NewColumn = OpponentsPieces[Index][COLUMN]
  OpponentsPieces[Index][ROW] = -1
  OpponentsPieces[Index][COLUMN] = -1
  return NewRow, NewColumn, OpponentsPieces

def MovePiece(Board, PlayersPieces, OpponentsPieces,
ChosenPiece, NewRow, NewColumn):
  Index = int(ChosenPiece[1:])
  CurrentRow = PlayersPieces[Index][ROW]
  CurrentColumn = PlayersPieces[Index][COLUMN]
  Board[CurrentRow][CurrentColumn] = SPACE

  if NewRow == BOARD_SIZE - 1 and
PlayersPieces[Index][DAME] == 0:
    Player = 'a'
    PlayersPieces[0][1] += 1
    PlayersPieces[Index][DAME] = 1
    ChosenPiece = ChosenPiece.upper()
    NewRow, NewColumn, OpponentsPieces = MoveDame(Player,
OpponentsPieces)
  elif NewRow == 0 and PlayersPieces[Index][DAME] == 0:
    Player = 'b'
    PlayersPieces[0][1] += 1
    PlayersPieces[Index][DAME] = 1
    ChosenPiece = ChosenPiece.upper()
    NewRow, NewColumn, OpponentsPieces = MoveDame(Player,
OpponentsPieces)
  PlayersPieces[Index][ROW] = NewRow
  PlayersPieces[Index][COLUMN] = NewColumn
  Board[NewRow][NewColumn] = ChosenPiece
  return Board, PlayersPieces, OpponentsPieces
```

```
def MakeMove(Board, PlayersPieces, OpponentsPieces,
ListOfMoves, PieceIndex):
  PlayersPieces[0][0] += 1
  if PieceIndex > 0:
    Piece = ListOfMoves[PieceIndex].Piece
    NewRow = ListOfMoves[PieceIndex].NewRow
    NewColumn = ListOfMoves[PieceIndex].NewColumn
    PlayersPieceIndex = int(Piece[1:])
    CurrentRow = PlayersPieces[PlayersPieceIndex][ROW]
    CurrentColumn =
PlayersPieces[PlayersPieceIndex][COLUMN]
    Jumping = ListOfMoves[PieceIndex].CanJump
    Board, PlayersPieces, OpponentsPieces =
MovePiece(Board, PlayersPieces, OpponentsPieces, Piece,
NewRow, NewColumn)
    if Jumping:
      MiddlePieceRow = (CurrentRow + NewRow) // 2
      MiddlePieceColumn = (CurrentColumn + NewColumn) // 2
      MiddlePiece =
Board[MiddlePieceRow][MiddlePieceColumn]
      print 'jumped over ', MiddlePiece
  return Board, PlayersPieces, OpponentsPieces
```

Pascal

03 1 **11**

```
var
  NumberIn, NumberOut, Count, PartValue, i: integer;
begin
  write('Enter a positive whole number: ');
  readln(NumberIn);
  NumberOut := 0;
  Count := 0;
  while NumberIn > 0 do
    begin
      Count := Count + 1;
      PartValue := NumberIn mod 2;
      NumberIn := NumberIn div 2;
      for i := 1 to Count - 1 do
        PartValue := PartValue * 10;
      NumberOut := NumberOut + PartValue;
    end;
  writeln('The result is: ', NumberOut);
end;
```

14 1 **3**

```
procedure DisplayErrorCode(ErrorNumber: integer);
begin
  write('Error Code ', ErrorNumber, ' - ');
  case ErrorNumber of
  1 : writeln('not a valid piece');
  2 : writeln('not a valid move');
  3 : writeln('not a number');
  4 : writeln('file error');
  end;
end;
```

15 1 **2**

```
function ValidJump(Board: TBoard; PlayersPieces: TPieces;
Piece: string; NewRow, NewColumn: integer): boolean;
var
  Valid: boolean;
  MiddlePiece: string;
  Player, OppositePiecePlayer, MiddlePiecePlayer: string;
  Index, CurrentRow, CurrentColumn, MiddlePieceRow,
MiddlePieceColumn: integer;
begin
  Valid := false;
  MiddlePiece := '';
  Player := LowerCase(LeftStr(Piece, 1));
  if Length(Piece) = 2 then
    Index := StrtoInt(RightStr(Piece, 1))
  else
    Index := StrtoInt(RightStr(Piece, 2));
  if Player = 'a' then
    OppositePiecePlayer := 'b'
  else
    OppositePiecePlayer := 'a';
  if (NewRow >= 0) and (NewRow < BoardSize)
    and (NewColumn >= 0) and (NewColumn < BoardSize) then
    if Board[NewRow, NewColumn] = Space then
      begin
        CurrentRow := PlayersPieces[Index, Row];
        CurrentColumn := PlayersPieces[Index, Column];
        MiddlePieceRow := (CurrentRow + NewRow) div 2;
        MiddlePieceColumn := (CurrentColumn + NewColumn)
div 2;
        MiddlePiece := Board[MiddlePieceRow,
MiddlePieceColumn];
        MiddlePiecePlayer :=
LowerCase(LeftStr(MiddlePiece, 1));
        if (MiddlePiecePlayer = OppositePiecePlayer) then
          Valid := true;
      end;
  ValidJump := Valid;
end;
```

Alternative logic statement:

```
(MiddlePiecePlayer = OppositePiecePlayer) and
(MiddlePiecePlayer <> ' ')
```

16 1 **9**

```
function CountNumberOfPieces(PlayersPieces: TPieces):
integer;
var Count, Index: integer;
begin
  Count := 0;
  for Index := 1 to NumberOfPieces do
    if PlayersPieces[Index, ROW] > -1 then
// allow Column instead of Row
      Count := Count + 1;
  CountNumberOfPieces := Count;
end;

procedure PrintResult(A, B: TPieces; NextPlayer: string);
var TotalA, TotalB: integer;
begin
  writeln('Game ended');
  TotalA := CountNumberOfPieces(A);
  TotalB := CountNumberOfPieces(B);
  TotalA := A[0, 0] - TotalA - 10 * A[0, 1];
  TotalB := B[0, 0] - TotalB - 10 * B[0, 1] ;
  if TotalA < TotalB then
    begin
      writeln('A won this game with a score of ', TotalA);
      writeln('B got a score of ', TotalB);
    end
  else
    if TotalB < TotalA then
      begin
        writeln('B won this game with a score of ',
TotalB);
        writeln('A got a score of ', TotalA);
      end
    else
      writeLn('it was a draw. Both players got a score of
', TotalA);
  PrintPlayerPieces(A, B);
end;
```

17	2	(code below)	9

```
procedure MoveDame(Player: string; var OpponentsPieces:
TPieces; var NewRow, NewColumn: integer);
var
  Opponent, ChosenPiece: string;
  Index: integer;
begin
  NewRow := -1;
  Opponent := '';
  while (Player = Opponent) or (NewRow = -1) do
    begin
      write('Which piece do you want to take? ');
      readln(ChosenPiece);
      Opponent := LowerCase(LeftStr(ChosenPiece,1));
      if Length(ChosenPiece) = 2 then
        Index := StrtoInt(RightStr(ChosenPiece,1))
      else
        Index := StrtoInt(RightStr(ChosenPiece,2));
      NewRow := OpponentsPieces[Index, Row];
      NewColumn := OpponentsPieces[Index][Column];
    end;
  OpponentsPieces[Index, Row] := -1;
  OpponentsPieces[Index, Column] := -1;
end;

procedure MakeMove(var Board: TBoard; var PlayersPieces,
  OpponentsPieces: TPieces; ListOfMoves: TList;
PieceIndex: integer);
var
  Piece, MiddlePiece: string;
  NewRow, NewColumn, PlayersPieceIndex, CurrentRow,
CurrentColumn: integer;
  MiddlePieceRow, MiddlePieceColumn: integer;
  Jumping: boolean;
begin
  PlayersPieces[0, 0] := PlayersPieces[0, 0] + 1;
  if PieceIndex > 0 then
    begin
      Piece := ListOfMoves[PieceIndex].Piece;
      NewRow := ListOfMoves[PieceIndex].NewRow;
      NewColumn := ListOfMoves[PieceIndex].NewColumn;
      if Length(Piece) = 2 then
        PlayersPieceIndex := StrtoInt(RightStr(Piece, 1))
      else
        PlayersPieceIndex := StrtoInt(RightStr(Piece, 2));
      CurrentRow := PlayersPieces[PlayersPieceIndex, Row];
      CurrentColumn := PlayersPieces[PlayersPieceIndex,
Column];
      Jumping := ListOfMoves[PieceIndex].CanJump;
      MovePiece(Board, PlayersPieces, OpponentsPieces,
Piece, NewRow, NewColumn);
      if Jumping then
        begin
          MiddlePieceRow := (CurrentRow + NewRow) div 2;
          MiddlePieceColumn := (CurrentColumn + NewColumn)
div 2;
          MiddlePiece := Board[MiddlePieceRow,
MiddlePieceColumn];
        end;
    end;
end;

procedure MovePiece(var Board: TBoard; var PlayersPieces,
OpponentsPieces: TPieces;
  ChosenPiece: string; NewRow, NewColumn: integer);
var
  Index, CurrentRow, CurrentColumn: integer;
  Player: string;
begin
  if Length(ChosenPiece) = 2 then
    Index := StrtoInt(RightStr(ChosenPiece,1))
  else
    Index := StrtoInt(RightStr(ChosenPiece,2));
  CurrentRow := PlayersPieces[Index, Row];
  CurrentColumn := PlayersPieces[Index, Column];
  Board[CurrentRow, CurrentColumn] := Space;

  if (NewRow = BoardSize-1) and (PlayersPieces[Index,
Dame] = 0) then
    begin
      Player := 'a';
      PlayersPieces[0,1] := PlayersPieces[0,1] + 1;
      PlayersPieces[Index, Dame] := 1;
      ChosenPiece := UpperCase(ChosenPiece);
      MoveDame(Player, OpponentsPieces, NewRow,
NewColumn);
    end
  else
    if (NewRow = 0) and (PlayersPieces[Index, Dame] = 0)
then
      begin
        Player := 'b';
        PlayersPieces[0, 1] := PlayersPieces[0, 1] + 1;
        PlayersPieces[Index, Dame] := 1;
        ChosenPiece := UpperCase(ChosenPiece);
        MoveDame(Player, OpponentsPieces, NewRow,
NewColumn);
      end;
  PlayersPieces[Index, Row] := NewRow;
  PlayersPieces[Index, Column] := NewColumn;
  Board[NewRow, NewColumn] := ChosenPiece;
end;

procedure MakeMove(var Board: TBoard; var PlayersPieces,
  OpponentsPieces: TPieces; ListOfMoves: TList;
PieceIndex: integer);
var
  Piece, MiddlePiece: string;
  NewRow, NewColumn, PlayersPieceIndex, CurrentRow,
CurrentColumn: integer;
  MiddlePieceRow, MiddlePieceColumn: integer;
  Jumping: boolean;
begin
```

```
  PlayersPieces[0, 0] := PlayersPieces[0, 0] + 1;
  if PieceIndex > 0 then
    begin
      Piece := ListOfMoves[PieceIndex].Piece;
      NewRow := ListOfMoves[PieceIndex].NewRow;
      NewColumn := ListOfMoves[PieceIndex].NewColumn;
      if Length(Piece) = 2 then
        PlayersPieceIndex := StrtoInt(RightStr(Piece, 1))
      else
        PlayersPieceIndex := StrtoInt(RightStr(Piece, 2));
      CurrentRow := PlayersPieces[PlayersPieceIndex, Row];
      CurrentColumn := PlayersPieces[PlayersPieceIndex,
Column];
      Jumping := ListOfMoves[PieceIndex].CanJump;
      MovePiece(Board, PlayersPieces, OpponentsPieces,
Piece, NewRow, NewColumn);
      if Jumping then
        begin
          MiddlePieceRow := (CurrentRow + NewRow) div 2;
          MiddlePieceColumn := (CurrentColumn + NewColumn)
div 2;
          MiddlePiece := Board[MiddlePieceRow,
MiddlePieceColumn];
        end;
    end;
end;
```

C#

03	1	
		```
		int count = 0, partValue, numberIn, numberOut = 0;
		Console.Write("Enter a positive whole number: ");
		numberIn = Convert.ToInt32(Console.ReadLine());
		while (numberIn > 0)
		{
		    count++;
		    partValue = numberIn % 2;
		    numberIn = numberIn / 2;
		    for (int i = 1; i < count; i++)
		    {
		        partValue = partValue * 10;
		    }
		    numberOut = numberOut + partValue;
		}
		Console.WriteLine("The result is: " + numberOut );
		Console.ReadLine();
		```
			11

14	1	
		```
		private static void DisplayErrorCode(int errorNumber)
		{
		    Console.WriteLine("Error Code " + errorNumber);
		    if (errorNumber == 1)
		    {
		        Console.WriteLine("not a valid piece");
		    }
		    else if (errorNumber == 2)
		    {
		        Console.WriteLine("not a valid move");
		    }
		    else if (errorNumber == 3)
		    {
		        Console.WriteLine("not a number");
		    }
		    else if (errorNumber == 4)
		    {
		        Console.WriteLine("file error");
		    }
		}
		```
			3

15	1	
		```
		private static bool ValidJump(string[,] board, int[,]
		playersPieces, string piece, int newRow, int newColumn)
		{
		    string middlePiece = "";
		    string player, oppositePiecePlayer, middlePiecePlayer;
		    int index, currentRow, currentColumn, middlePieceRow,
		middlePieceColumn;
		    player = piece[0].ToString().ToLower();
		    if (piece.Length == 2)
		    {
		        index = Convert.ToInt32(piece[1].ToString());
		    }
		    else
		    {
		        index = Convert.ToInt32(piece.Substring(1));
		    }
		    if (player == "a")
		```
			2

```
    {
        oppositePiecePlayer = "b";
    }
    else
    {
        oppositePiecePlayer = "a";
    }
    if (newRow >= 0 && newRow < BoardSize &&
        newColumn >= 0 && newColumn < BoardSize)
    {
        if (board[newRow, newColumn] == Space)
        {
            currentRow = playersPieces[index, Row];
            currentColumn = playersPieces[index, Column];
            middlePieceRow = (currentRow + newRow) / 2;
            middlePieceColumn = (currentColumn +
newColumn) / 2;
            middlePiece = board[middlePieceRow,
middlePieceColumn];
            middlePiecePlayer =
middlePiece[0].ToString().ToLower();
            if (middlePiecePlayer == oppositePiecePlayer)
            {
                return true;
            }
        }
    }
    return false;
}
```

Alternative logic statement:

```
(middlePiecePlayer == oppositePiecePlayer) &&
middlePiecePlayer != " "
```

16	1	(see code below)	**9**

```
private static int CountNumberOfPieces(int[,]
playersPieces)
{
    int count = 0;
    for (int index = 1; index < NumberOfPieces + 1;
index++)
    {
        if (playersPieces[index,Row] > -1) // allow Column
instead of Row
        {
            count++;
        }
    }
    return count;
}

private static void PrintResult(int[,] a, int[,] b, string
nextPlayer)
{
    int totalA, totalB;
    Console.WriteLine("Game ended");
    totalA = CountNumberOfPieces(a);
    totalB = CountNumberOfPieces(b);
    totalA = a[0, 0] - totalA - 10 * a[0, 1];
    totalB = b[0, 0] - totalB - 10 * b[0, 1];
    if (totalA < totalB)
    {
        Console.WriteLine("A won this game with a score of
" + totalA);
        Console.WriteLine("B got a score of " + totalB);
    }
    else if (totalB < totalA)
    {
        Console.WriteLine("B won this game with a score of
" + totalB);
        Console.WriteLine("A got a score of " + totalA);
    }
    else
    {
        Console.WriteLine("it was a draw. Both players got
a score of " + totalA);
    }
    PrintPlayerPieces(a, b);
}
```

17	2	(code below)	9

```
private static void MoveDame(string[,] board, string
player, ref int newRow,
    ref int newColumn, int[,] opponentsPieces)
{
    string opponent, chosenPiece;
    int index = 0;
    newRow = -1;
    opponent = "";
    while ((player == opponent) || (newRow == -1))
    {
        Console.Write("Which piece do you want to take?
");
        chosenPiece = Console.ReadLine();
        opponent = chosenPiece[0].ToString().ToLower();
        index = Convert.ToInt32(chosenPiece.Substring(1));
        newRow = opponentsPieces[index, Row];
        newColumn = opponentsPieces[index, Column];
    }
    opponentsPieces[index, Row] = -1;
    opponentsPieces[index, Column] = -1;
}
private static void MovePiece(string[,] board, int[,]
playersPieces,
    string chosenPiece, int newRow, int newColumn, int[,]
opponentsPieces)
{
    int index, currentRow, currentColumn;
    string player;
    if (chosenPiece.Length == 2)
    {
        index =
Convert.ToInt32(chosenPiece[1].ToString());
    }
    else
    {
        index = Convert.ToInt32(chosenPiece.Substring(1));
    }
    currentRow = playersPieces[index, Row];
    currentColumn = playersPieces[index, Column];
    board[currentRow, currentColumn] = Space;
    if (newRow == BoardSize - 1 && playersPieces[index,
Dame] == 0)
    {
        player = "a";
        playersPieces[0, 1] = playersPieces[0, 1] + 1;
        playersPieces[index, Dame] = 1;
        chosenPiece = chosenPiece.ToUpper();
        MoveDame(board, player, ref newRow, ref newColumn,
opponentsPieces);
    }
    else if (newRow == 0 && playersPieces[index, Dame] ==
0)
    {
        player = "b";
        playersPieces[0, 1] = playersPieces[0, 1] + 1;
        playersPieces[index, Dame] = 1;
        chosenPiece = chosenPiece.ToUpper();
        MoveDame(board, player, ref newRow, ref newColumn,
opponentsPieces);
    }
    playersPieces[index, Row] = newRow;
    playersPieces[index, Column] = newColumn;
    board[newRow, newColumn] = chosenPiece;
}

  private static void MakeMove(string[,] board, int[,]
playersPieces, int[,] opponentsPieces, MoveRecord[]
listOfMoves, int pieceIndex)
    {
      string piece, middlePiece;
      int newRow, newColumn, playersPieceIndex,
currentRow, currentColumn;
      int middlePieceRow, middlePieceColumn;
      bool jumping;
      playersPieces[0, 0] = playersPieces[0, 0] + 1;
      if (pieceIndex > 0)
      {
        piece = listOfMoves[pieceIndex].Piece;
        newRow = listOfMoves[pieceIndex].NewRow;
        newColumn = listOfMoves[pieceIndex].NewColumn;
        playersPieceIndex =
Convert.ToInt32(piece.Substring(1));
        currentRow = playersPieces[playersPieceIndex,
Row];
        currentColumn = playersPieces[playersPieceIndex,
Column];
        jumping = listOfMoves[pieceIndex].CanJump;
        MovePiece(board, playersPieces, piece, newRow,
newColumn, opponentsPieces);
```

		```
		        if (jumping)
		        {
		          middlePieceRow = (currentRow + newRow) / 2;
		          middlePieceColumn = (currentColumn + newColumn)
		/ 2;
		          middlePiece = board[middlePieceRow,
		middlePieceColumn];
		          Console.WriteLine("jumped over " + middlePiece);
		        }
		      }
		    }
		```

Java

03 | 1 | **11**

```
Console.writeLine("Enter a positive whole number: ");
int numberIn = Integer.parseInt(Console.readLine());
int numberOut = 0;
int count = 0;
int partValue;
while (numberIn > 0) {
    count++;
    partValue = numberIn % 2;
    numberIn = numberIn / 2;
    for (int i = 1; i < count; i++) {
        partValue = partValue * 10;
    }
    numberOut = numberOut + partValue;
}
Console.writeLine("The result is: " + numberOut);
```

14 | 1 | **3**

```
void displayErrorCode(int errorNumber) {
    Console.write("Error Code " + errorNumber + " - ");
    if (errorNumber == 1) {
        Console.writeLine("not a valid piece");
    } else if (errorNumber == 2) {
        Console.writeLine("not a valid move");
    } else if (errorNumber == 3) {
        Console.writeLine("not a number");
    } else if (errorNumber == 4) {
        Console.writeLine("file error");
    }
}
```

Alternative Example

```
void displayErrorCode(int errorNumber) {
    Console.write("Error " + errorNumber + " - ");
    switch (errorNumber) {
        case 1:
            Console.writeLine("not a valid piece.");
            break;
        case 2:
            Console.writeLine("not a valid move");
            break;
        case 3:
            Console.writeLine("not a number");
            break;
        case 4:
            Console.writeLine("file error");
            break;
    }
}
```

15 | 1 | **2**

```
boolean validJump(String[][] board, int[][] playersPieces,
String piece, int newRow, int newColumn) {
    boolean valid = false;
    String oppositePiecePlayer, middlePiecePlayer, player,
middlePiece;
    int index, currentRow, currentColumn, middlePieceRow,
middlePieceColumn;
    player = (piece.charAt(0) + "").toLowerCase();
    index = Integer.parseInt(piece.substring(1));
    if (player.equals("a")) {
        oppositePiecePlayer = "b";
    } else {
        oppositePiecePlayer = "a";
    }
    if (newRow >= 0 && newRow < BOARD_SIZE
            && newColumn >= 0 && newColumn < BOARD_SIZE) {
        if (board[newRow][newColumn].equals(SPACE)) {
            currentRow = playersPieces[index][ROW];
            currentColumn = playersPieces[index][COLUMN];
            middlePieceRow = (currentRow + newRow) / 2;
            middlePieceColumn = (currentColumn +
newColumn) / 2;
            middlePiece =
board[middlePieceRow][middlePieceColumn];
            middlePiecePlayer = (middlePiece.charAt(0) +
"").toLowerCase();
            if
(middlePiecePlayer.equals(oppositePiecePlayer)) {
                valid = true;
            }
        }
    }
    return valid;
}
```

16	1	(code below)	9

```
int countNumberOfPieces(int[][] playerPieces) {
    int count = 0;
    for (int index = 1; index < NUMBER_OF_PIECES + 1;
index++) {
        if (playerPieces[index][ROW] > -1) {
            count++;
        }
    }
    return count;
}

void printResult(int[][] a, int[][] b, String nextPlayer)
{
    Console.writeLine("Game ended");
    int totalA = countNumberOfPieces(a);
    int totalB = countNumberOfPieces(b);
    totalA = a[0][0] - totalA - 10 * a[0][1];
    totalB = b[0][0] - totalB - 10 * b[0][1];
    if (totalA < totalB) {
        Console.writeLine("A won with a score of " +
totalA);
        Console.writeLine("B got a score of " + totalB);
    } else if (totalB < totalA) {
        Console.writeLine("B won with a score of " +
totalB);
        Console.writeLine("A got a score of " + totalA);
    } else {
        Console.writeLine("it was a draw. Both players got
a score of " + totalA);
    }
    printPlayerPieces(a, b);
}
```

17	2		**9**

```
int[] moveDame(String player, int [][] opponentsPieces) {
    int newRow = -1;
    int newColumn = 0;
    String opponent = "";
    int index = 0;
    while (player.equals(opponent) || newRow == -1) {
        Console.writeLine("Which piece do you want to
take?");
        String chosenPiece = Console.readLine();
        opponent = chosenPiece.substring(0,
1).toLowerCase();
        index =
Integer.parseInt(chosenPiece.substring(1));
        newRow = opponentsPieces[index][ROW];
        newColumn = opponentsPieces[index][COLUMN];
    }
    opponentsPieces[index][ROW] = -1;
    opponentsPieces[index][COLUMN] = -1;
    return new int[]{newRow, newColumn};
}

void movePiece(String[][] board, int[][] playersPieces,
int[][] opponentsPieces, String chosenPiece, int newRow,
int newColumn) {
    int index =
Integer.parseInt(chosenPiece.substring(1));
    int currentRow = playersPieces[index][ROW];
    int currentColumn = playersPieces[index][COLUMN];
    board[currentRow][currentColumn] = SPACE;
    String player;

    if (newRow == BOARD_SIZE - 1 &&
playersPieces[index][DAME] == 0) {
        player = "a";
        playersPieces[0][1] += 1;
        playersPieces[index][DAME] = 1;
        chosenPiece = chosenPiece.toUpperCase();
        int[] rtnInts = moveDame(player, opponentsPieces);
        newRow = rtnInts[0];
        newColumn = rtnInts[1];
    } else if (newRow == 0 && playersPieces[index][DAME]
== 0) {
        player = "b";
        playersPieces[0][1] += 1;
        playersPieces[index][DAME] = 1;
        chosenPiece = chosenPiece.toUpperCase();
        int[] rtnInts = moveDame(player, opponentsPieces);
        newRow = rtnInts[0];
        newColumn = rtnInts[1];
    }
    playersPieces[index][ROW] = newRow;
    playersPieces[index][COLUMN] = newColumn;
    board[newRow][newColumn] = chosenPiece;
}

void makeMove(String[][] board, int[][] playersPieces,
        int[][] opponentsPieces, MoveRecord[] listOfMoves,
int pieceIndex) {
    playersPieces[0][0] += 1;
    if (pieceIndex > 0) {
        String piece = listOfMoves[pieceIndex].piece;
        int newRow = listOfMoves[pieceIndex].newRow;
        int newColumn = listOfMoves[pieceIndex].newColumn;
        int playersPieceIndex =
Integer.parseInt(piece.substring(1));
        int currentRow =
playersPieces[playersPieceIndex][ROW];
        int currentColumn =
playersPieces[playersPieceIndex][COLUMN];
        boolean jumping = listOfMoves[pieceIndex].canJump;
        movePiece(board, playersPieces, opponentsPieces,
piece, newRow, newColumn);
        if (jumping) {
            int middlePieceRow = (currentRow + newRow) /
2;
            int middlePieceColumn = (currentColumn +
newColumn) / 2;
            String middlePiece =
board[middlePieceRow][middlePieceColumn];
            Console.writeLine("jumped over " +
middlePiece);
        }
    }
}
```