

**Programming Tasks Part 2**

These questions require you to load the **Skeleton Program** and to make programming changes to it.

*Note that any alternative or additional code changes that you deemed appropriate to make must also be evidenced   
– ensuring that it is clear where in the Skeleton Program those changes have been made.*

# Task 1

# Task 11

**Task 11** **Marks:** 11

This question refers to the **PlayGame** method together with a new method called **ModifyQueueOptions** in the **Dastan** class, additional new methods **ReverseQueue**, **SwapFirstAndLast** and **MoveItemToFront** in the **MoveOptionQueue** class together with new methods **ReplaceQueue**, **GetMoveOptionQueue**, **ReversePlayerQueue**, **SwapFirstAndLast** and **MoveItemToFront** in the **Player** class.

Introduce a new option 6 to the main game playing menu. On selecting this option, a player can choose sub options for making changes to their move queue using the following menu:

|  |
| --- |
| **Options** |
| 1. Reverse the current player queue |
| 1. Swap the current player queue with the opponent queue |
| 1. Swap the first and last elements in the current player queue |
| 1. Move one of the move options to the front of the current player queue |
| 1. Nothing (make normal move) |

**Note:** Options 1–4 cost 3 points, but the player can choose option 5 for free.

**Note:** This does not count as the player’s turn and the player should still be able to play a move.

**What you need to do**

**Task 1**

Modify the **Dastan** class to introduce the new menu option.

1. Modify the **PlayGame** method to add option 6 to the move options menu.
2. Create a new private method in the **Dastan** class called **ModifyQueueOptions** which gives the player the above menu. Include validation to ensure that the user can only enter one of the option choices from the menu.
3. Adjust the score by 3 if options 1–4 are chosen but not if option 5 is.

**Task 2**

Modify the **MoveOptionQueue** class to add the required methods.

1. Create new method **ReverseQueue** to allow the current player’s queue to be reversed.
2. Create new method **SwapFirstAndLast** to swap the first and last elements of the current player’s queue.
3. Create new method **MoveItemToFront** to move the item from the chosen position to the start of the queue for the current player. There is no need to validate the input for the position to move the option from.

**(TASK CONTINUES ON THE NEXT PAGE)**

**Task 3**

Modify the Player class to create the required methods.

1. Create new methods **ReversePlayerQueue**, **SwapFirstAndLast**, **MoveItemToFront** in the **Player** class to expose the new **MoveQueueOptions** choices/methods to the **Dastan** class.
2. Create a new method **ReplaceQueue** to allow the current player’s queue to be replaced with the queue passed in as a parameter.
3. Create a new method GetMoveOptionQueue which returns the player’s queue to be used in the newly created ReplaceQueue method.

**Task 4**

Test that the changes you have made work:

* run the skeleton program.
* show player one selecting option 6 from the main game menu.
* show the player selecting each one of the queue options in turn and the updated queue on the screen as a result of the change.

**Evidence that you need to provide:**

* PROGRAM SOURCE CODE showing changes made to the **PlayGame** method
* PROGRAM SOURCE CODE for the new **ModifyQueueOptions** method in the **Dastan** class
* PROGRAM SOURCE CODE showing changes made to the **MoveOptionsQueue** class
* PROGRAM SOURCE CODE showing changes made to the **Player** class
* SCREEN CAPTURE(S) showing the required test

# Task 12

**Task 12** **Marks:** 7

This question refers to the creation of a new protected attribute **NoOfPieces**, modification of the existing **PlayGame** method and creation of two new methods **CheckReincarnation** and **CountNormalPieces** in the **Dastan** class.

Introduce a new feature whereby if a player manages to get one of their players to the opponent’s back row, the player is given a new piece to place on any unoccupied space on their own back row. Note that the player cannot reincarnate pieces that are not dead so they should not be able to have more pieces than they started with.

**What you need to do**

**Task 1**

Create a new private method in the **Dastan** class called **CountNormalPieces** that will return the number of pieces that the current player has excluding the Mirza.

**Task 2**

1. Modify the constructor of the **Dastan** class to store the number of pieces passed in as a new protected attribute called **NoOfPieces**.
2. Modify the **PlayGame** method of the **Dastan** class to call a new private method **CheckReincarnation** following a legal move.

**Task 3**

Create a new private method **CheckReincarnation** in the **Dastan** class. This should take one parameter which is the **FinishSquareReference** for the current player’s move. If the player’s move ended on the opponent’s back row (e.g. row 6 for player one) and the player has fewer pieces than they started with, then allow the player to reincarnate a piece on their back row in an empty square. You need to validate that the square is empty and allow the player to reselect if it is not.

**Task 4**

Test that the changes you have made work:

* add the following four lines of code to the START of the private method **CreatePieces** in the **Dastan** class: *(be certain to remove this after testing to ensure that the board returns to normal)*

NoOfPieces = 2;

Board[GetIndexOfSquare(51)].SetPiece(new Piece("piece", Players[0], 1, "!"));

Board[GetIndexOfSquare(21)].SetPiece(new Piece("piece", Players[1], 1, "\""));

Board[GetIndexOfSquare(54)].SetPiece(new Piece("piece", Players[1], 1, "\""));

* run the skeleton program.
* select a Ryott move for player one, enter a start square of 51 and an end square of 61.
* show player one attempting to reincarnate a piece in column 3 and being given an error message saying that the square must be empty.
* show player one attempting to reincarnate a piece in column 4 and the board being updated appropriately.
* select a Ryott move for player two, enter a start square of 21 and an end square of 11.
* show player two not receiving a reincarnation message.
* remove any modifications to the CreatePieces method after testing by removing the additional lines so that the board returns to normal.

**Evidence that you need to provide:**

* PROGRAM SOURCE CODE showing the new CountNormalPieces method in the Dastan class
* PROGRAM SOURCE CODE showing the new CheckReincarnation method in the Dastan class
* PROGRAM SOURCE CODE showing the other code changes to the Dastan class
* SCREEN CAPTURE(S) showing the required test

# ask 13

**Task 13** **Marks:** 8

This question refers to the **PlayGame** method together with modification of the **CreateBoard** method in the **Dastan** class. Additionally it involves the creation of a new method **GetCampedTwoTurns** in the **Square** class and the creation of a new **Taziz** class (Reinforcement) which inherits from **Square**.

Create a new type of game square, the Ta’ziz (similar to the Kotla), which is placed in the middle of the playing board (or slightly closer to player one if there are an even number of rows). Placement of the Ta’ziz should use the NoOfRows and NoOfColums attributes to ensure that it is placed in the middle of the board, regardless of the size of the board. Either player can occupy the Ta’ziz with any of their pieces. Two players cannot both occupy the Ta’ziz simultaneously. A second player entering the Ta’ziz will remove the first player piece from the board and takes ownership of the square just like any other square on the board. If a player occupies the Ta’ziz for two turns by both players without being captured by their opponent (entering the Ta’ziz is considered a player’s first turn), then their next move choice will have zero cost. This gives a player a zero cost move, but risks sitting in the middle of the playing board to get it. Continuing to camp in the Ta’ziz after two turns gains no further advantage.

**What you need to do**

**Task 1**

Create a new class called **Taziz** which should inherit from the **Square** class.

1. Add a new private attribute **CampedTurns** and initialise it to 0.
2. Override the **SetPiece** and **RemovePiece** methods from the **Square** class. **SetPiece** should adjust the **Taziz** symbol to an upper case ‘A’ if player one owns the Ta’ziz and a lower case ‘a’ if player two owns the Ta’ziz (you may assume that the player with a **Direction** of 1 is the player at the top – player one). When a player piece leaves the Ta’ziz, ownership of the square should be set to null and the symbol set to a lower case ‘x’. Each time the Ta’ziz is captured by a new player **CampedTurns** should be reset back to zero.
3. Override the new method **GetCampedTwoTurns** from the **Square** class. The **GetCampedTwoTurns** method should check the number of turns using the **CampedTurns** attribute and return true if it is = 4.

**Task 2**

Modify the **CreateBoard** method in the **Dastan** class to place an Ta’ziz on the square closest to the middle of the board with a lower case ‘x’ symbol when the board is first created.

***NOTE:*** *The Ta’ziz* *should be correctly placed on the board even if the size is not the original 6x6, i.e. it should take account of the number of columns and rows.*

In the case where there are an even number of rows, the castle should be slightly closer to player one; also if there are an even number of columns then it should be slightly closer to the left. In the case of the starting board this will place it on square 33, but it should work for **any size board**.

The initial Ta’ziz does not belong to either player.

**(TASK CONTINUES ON THE NEXT PAGE)**

**Task 3**

Modify the **PlayGame** method so that if a move is legal the game should test to see if the Ta’ziz has been camped in for two full turns and, if so, give the selected move to the player at zero cost.

**Task 4**

Test that the changes you have made work:

* run the skeleton program.
* use a Cuirassier move option 3 to move a player one piece into the Ta’ziz (from 23 to 33).
* play the game until both players have had two turns – leaving the player one piece in the Ta’ziz without attacking it using player two.
* after both players have had two turns, show a move option by player one which incurs zero cost.

**Evidence that you need to provide:**

* PROGRAM SOURCE CODE showing changes made to the **PlayGame** method
* PROGRAM SOURCE CODE showing changes made to the **CreateBoard** method
* PROGRAM SOURCE CODE showing the new **GetCampedTwoTurns** virtual method in the **Square** class
* PROGRAM SOURCE CODE showing the new **Taziz** class
* SCREEN CAPTURE(S) showing the required test

# Task 14

# Task 15

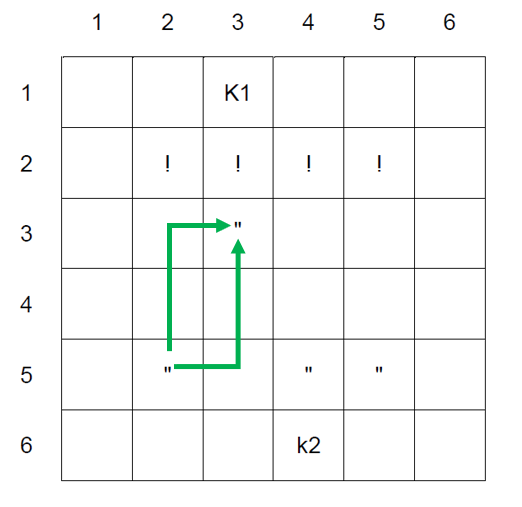
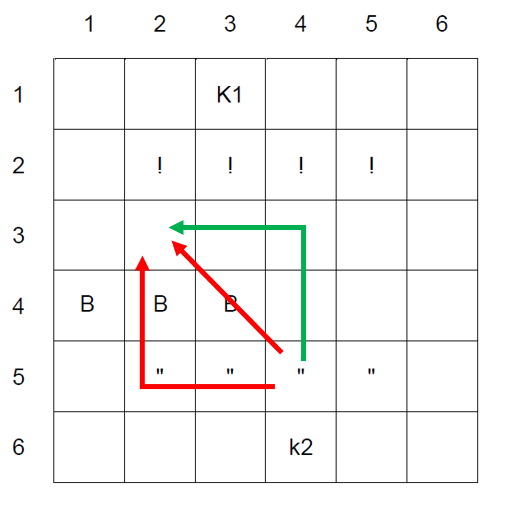
**Task 15** **Marks:** 15

This question refers to the **PlayGame** method together with modification of the **CheckSquareIsValid** and **CreatePieces** methods and creation of three new private methods, **CheckBarrierIsValid**, **PlaceBarrier** and **CheckManhattanDistance** in the **Dastan** class. Additionally it involves the creation of new public method **ContainsBarrier** in the **Square** class and the creation of a new **Barrier** class which inherits from **Square**.

Create a new game piece called a Barrier. On creation of the board each player can choose where they would like to place their Barrier on the board. The Barrier is 3 squares wide. This cannot be outside of the board or in a position occupied by a normal piece or an opponent’s Barrier. The Barrier piece cannot be moved, occupied or jumped over by either player.

Some moves, however, do not move in a straight line, for example the Jazair. As shown in **Fig 1** below, the direct move would be through the Barrier which is not allowed. A move around the side and top of the Barrier, however, is possible which is, therefore allowed. Use the Manhattan distance to check if there is a move route possible around the edge of the Barrier.

Manhattan distance is a heuristic function for calculating distance between two locations, for example in a grid. In the case of Dastan it is calculated by counting the sum of the number of squares horizontally and then vertically (or vice versa) between a player starting location and the finishing location as shown in **Fig 2** below.



**Fig 1 Fig 2**

**What you need to do**

**Task 1**

1. Create a new class **Barrier** which should inherit from the **Square** class. A Barrier should be assigned an owner and given the symbol of a capital ‘B’ if it belongs to player one and a lowercase ‘b’ if it belongs to player two.
2. Create a new public method **ContainsBarrier** in the **Square** class which returns true if a Barrier has been placed in that square.

**(TASK CONTINUES ON THE NEXT PAGE)**

**Task 2**

1. Modify the **CheckSquareIsValid** method to check if the square being tested contains a Barrier so that a piece cannot occupy it or attempt to move it.
2. Create a new method **CheckBarrierIsValid** in the **Dastan** class which checks that the location of a Barrier being placed by a player fits within the bounds of the board and only covers empty squares.
3. Create a new method called **PlaceBarrier** in the **Dastan** class which places a three-square wide Barrier onto the board. The Barrier will always be horizontal and the player should enter the centre square when being asked where to place the Barrier.

**Task 3**

1. Create a new method called **CheckManhattanDistance** in the **Dastan** class which checks both paths from a starting square reference to a finishing square reference by traversing along the starting row then down the finishing column and also down the starting column and along the finishing row. This is used to check if a selected move can traverse around a Barrier rather than over the top of it.
2. Modify **PlayGame** to call **CheckManhattanDistance** which should be combined **CheckPlayerMove** using a logical AND to set the value of the variable **MoveLegal**.

**Note:** This should be used for all moves even if they are too short to potentially jump a Barrier as they may be able to go round.

**Task 4**

Test that the changes you have made work:

* run the skeleton program.
* enter a position of 34 for the player one Barrier.
* enter a position of 42 for the player two Barrier.
* for player one: choose 9, then 1, then 1, then 24, then 46.
* for player two: choose 3, then 53, then 31.
* for player one: choose 2, then 25, then 45.
* for player two: choose 1, then 52, then 42, then 51.

**Evidence that you need to provide:**

* PROGRAM SOURCE CODE showing changes made to the **PlayGame** method
* PROGRAM SOURCE CODE showing changes made to the **CheckSquareIsValid** and **CreatePieces** methods in the **Dastan** class
* PROGRAM SOURCE CODE for the new private **CheckBarrierIsValid, PlaceBarrier** and **CheckManhattanDistance** methods in the **Dastan** class
* PROGRAM SOURCE CODE showing changes made to the **Square** class
* PROGRAM SOURCE CODE showing the new **Barrier** class
* SCREEN CAPTURE(S) showing the required test