

**Exam-style Questions (Mark Scheme)**

| **Q** | | **Suggested Solution** | **Marks** | **Marking Guidance** |
| --- | --- | --- | --- | --- |
| **1** | (a) | O(n2) | 1 mark | **A**: polynomial time |
| (b) | 1 mark for each point   * Relatively efficient for smaller input sizes * … however, as the input size grows, the completion time increases * The rate of change is constantly changing using a quadratic function… * … which means that it does not scale up well | 3 marks | MAX 3 |
| **2** | (a) | 1 mark for each point   * You may mistype/misspell one of them * … which could mean that the code develops a logic error | 2 marks |  |
| (b) | 1 mark for each point   * One possible solution would be to define them as constants (1 mark) * … which would mean that you would get an error with an undefined identifier before running the program | 2 marks |  |
| **3** | (a) | Every square in the board is treated as a square **[1 mark]** but some of them may be Kotlas (which inherit from Square)  **[1 mark]** so will end up calling the overridden method on the Kotla due to polymorphism because although treated as a Square it will behave as a Kotla **[1 mark]** | 3 marks |  | |
| **4** | (a) | Because the position of player one’s Kotla is determined by the number of columns DIV 2 which gives 3 **[1 mark]** and the position of player two’s Kotla is determined by the number of columns DIV 2 and then add 1 which gives 4 **[1 mark]** | 2 marks | **A**: any explanation that refers to integer division and then the same result + 1 for player 2 | |
| (b) | Change the calculation for player one **[1 mark]** to (NoOfColumns+1) DIV 2 **[1 mark]** which will round up for odd numbers **[1 mark]** but round down for even numbers **[1 mark]** | 4 marks |  | |
| **5** | (a) | As the Direction attribute is part of the Player class **[1 mark]** both of these methods could go modify the NewMoveOption when it is received in the AddToMoveOptionQueue and UpdateMoveOptionQueueWithOffer methods **[1 mark]** to modify each non-zero value for RowChange and ColumnChange by multiplying it by the Direction for the current player **[1 mark]** | 3 marks | **A:** Alternative solutions such as putting the move inside the player or adding a ModifyMoveOption method to the player as long as they remove the need for the Direction parameter and make sense. | |
| **6** | (a) | A queue is more appropriate because move options are added to the end of the queue but could not be added to the bottom of the stack as it is a LIFO structure **[1 mark]** and removed from the front of the queue because it is a FIFO structure **[1 mark]** | 2 marks | **A:** any answer which explains that stacks have items added to and removed from the same end which is not suitable | |

| **Q** | | **Suggested Solution** | **Marks** | **Marking Guidance** |
| --- | --- | --- | --- | --- |
| **6** | (b) | A circular queue would need a head variable **[1 mark]** and a tail variable **[1 mark]**…    … so that when an item is added to the queue, the rear pointer could be incremented or wrapped back around to 0 if it was greater than 4 **[1 mark]** and when an item is removed from the queue the head pointer could be increased or wrapped back around to 0 if it was greater than 4 **[1 mark]** | 4 marks | **A:** indices starting at 1 and ending at 5  **A:** a well-labelled diagram as shown on its own as 4 marks (with the head and tail labels and the idea of wrapping around in a circle)  **A:** solutions using MOD to do the “wrapping back around”  **R:** references to built in stack/queue classes. |
| **7** | (a) | Each square is referred to by a two-digit number, the method extracts the first digit using MOD, subtracts one **[1 mark]** and then multiplies it by number of columns **[1 mark]**, then extracts the second digit of the square reference using DIV, subtracts one and adds the two together. **[1 mark]** | 3 marks | **R:** simply writing out a pseudocode version of the code |
| **8** | (a) | One dimension could be the row **[1 mark]** and the second dimension could be the column **[1 mark]** | 2 marks | **A:** 2D arrays represent grids and the board is a grid |
| (b) | An array is static so the amount of memory used will not change and the board size is fixed so this is appropriate | 1 mark | **A:** arrays are more efficient than lists  **A:** Arrays can be more efficient in memory than lists as they are contiguous. |
| **9** | (a) | Metadata describes the data in a file **[1 mark]**  Possible examples (any sensible answer will do):   * Board size (resolution) * Number of pieces for each player | 2 marks | **A:** Metadata is data about the data  **R:** File size as this is not stored in the file |
| **10** | (a) | This is not polymorphism because each of the five methods creates a MoveOption object **[1 mark]** which is the same class but contains different data **[1 mark]**. In order to be polymorphism you need to have child classes being treated as their parent which is not the case here **[1 mark]**. | 2 marks | MAX 2 of the 3 available points |
| (b) | This is polymorphism because each of the five different MoveOption methods (e.g. ChowkidarMoveOption) for each move inherits from MoveOption and so can be treated as a MoveOption **[1 mark]** but will actually behave as themselves **[1 mark]** meaning that you could still have a collection of MoveOptions, all of which would actually be children of MoveOption **[1 mark]** | 2 marks | MAX 2 of the 3 available points |
| **11** | (a) | 1 mark for each point   * super() is used to refer to the base class object * And call a method within it | 2 marks |  | |
| (b) | Overriding | 1 mark | **A**: override | |
| (c) | 1 mark for each point   * To provide **multiple** implementations * of a method with the same name * by selecting which version to run based on the number and type of parameters passed * within the same class definition | 3 marks | **R**: In an inherited class | |

| **Q** | | **Suggested Solution** | | **Marks** | **Marking Guidance** |
| --- | --- | --- | --- | --- | --- |
| **12** | a) | | 1 mark for each point   * A priority queue has different points at which items can join the queue according to priority * They join at the back of the section according to their priority, almost like sub-queues * If there are no items queued in the correct priority section then they join the queue at the front of the next lower priority or at the back of the next higher * Items are still taken one at a time from the front of the entire queue and still join at the back of the appropriate sub-queue/section | 4 marks | **A**: any equivalent points as long as they express the idea that you can queue jump to the BACK of a section according to priority |
| **13** | (a) | | 1 mark for each point   * The name describes the purpose of the variable * which makes the code easier to read/understand/follow | 2 marks |  |
| **14** | (a) | | 1 mark for each point   * It can be accessed by children/subclasses * and from within the class itself | 2 marks |  |
| (b) | | It can be accessed from anywhere | 1 mark | **A**: from outside the class |
| (c) | | It can only be accessed from within the class | 1 mark | **A**: opposite, e.g. it cannot be accessed from outside the class |
|  | (d) | | They allow correct encapsulation **[1 mark]** of the class… which means that you can only interact with the class through the intended interface **[1 mark]**… but it still allows for direct access within the class where required **[1 mark]**.Also avoids exposing attributes and methods that are either dangerous to expose or unnecessary outside the class **[1 mark]**. | 3 marks | MAX 3 |
| **15** | (a) | | 1 mark for each point   * Integer division returns a whole number (and a remainder) * Floating point division returns a decimal value with a decimal point | 2 marks | **A**: Whole number  **A**: Fraction for decimal |
| (b) | | It has two values, true or false | 1 mark | **A**: yes or no  **A**: 1 or 0  **I**: capitalisation and spelling |