TARGET CLEAR

A LEVEL AQA

PRE-RELEASE 2025

Programming Tasks (Mark Scheme)

It should be noted that solutions to each problem may vary depending on the techniques selected by the candidate. Marks should be awarded for solutions which achieve the correct outcome but through different techniques than those shown here.

Task 1 (3 marks)

Coding

- Prompt the user if they would like to quit. **[1 mark]**
- Exit the application loop correctly without decrementing the score variable on that turn. **[1 mark]**

Teacher Notes:
Candidates should ensure that once the user has chosen to quit the game, the application should not decrement the score or call the UpdateTargets method.

Example Solution

Modification to the `PlayGame` method:

```
            while (!GameOver)
            {
                DisplayState(Targets, NumbersAllowed, Score);
                //CHANGE
                Console.Write("Enter an expression or enter QUIT to end the game: ");
                UserInput = Console.ReadLine();
                if (UserInput.ToUpper() == "QUIT")
                {
                    GameOver = true;
                }
                else
                {
                    Console.WriteLine();
                    if (CheckIfUserInputValid(UserInput))
                    {
                        UserInputInRPN = ConvertToRPN(UserInput);
                        if (CheckNumbersUsedAreAllInNumbersAllowed(NumbersAllowed, UserInputInRPN, MaxNumber))
```

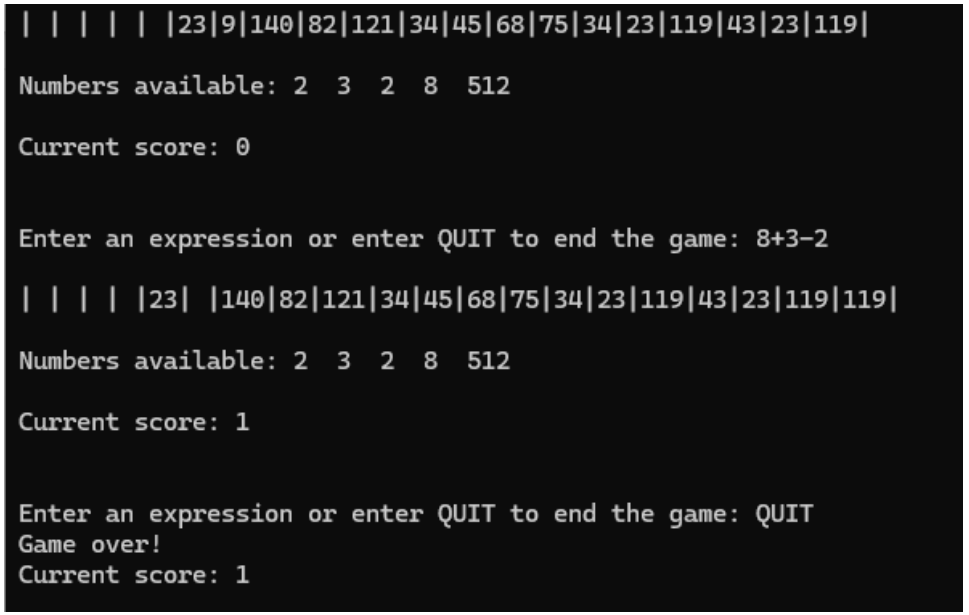


```
                    {
                        if (CheckIfUserInputEvaluationIsATarget(Targets, UserInputInRPN, ref Score))
                        {
                            RemoveNumbersUsed(UserInput, MaxNumber, NumbersAllowed);
                            NumbersAllowed = FillNumbers(NumbersAllowed, TrainingGame, MaxNumber);
                        }
                    }
                }
                Score--;
                if (Targets[0] != -1)
                {
                    GameOver = true;
                }
                else
                {
                    UpdateTargets(Targets, TrainingGame, MaxTarget);
                }
            }
        }
        //END CHANGE
        Console.WriteLine("Game over!");
        DisplayScore(Score);

    }
```

Testing

- Show the program displaying the identified target and final game score. **[1 mark]** ▶



Task 3 (2 marks)

Coding

- Refactor the code to remove the requirement for the Targets list to be iterated through. **[1 mark]**

Teacher Notes:
The objective of this question is to assess whether the candidate understands the functionality of a queue rather than iterating through a list.

Example Solution

Changes to `UpdateTargets`:

```
        static void UpdateTargets(List<int> Targets, bool TrainingGame, int MaxTarget)
        {
            //CHANGE
            Targets.RemoveAt(0);
            if (TrainingGame)
            {
                Targets.Add(Targets[Targets.Count - 1]);
            }
            else
            {
                Targets.Add(GetTarget(MaxTarget));
            }
            //END CHANGE
        }
```

Testing

- Show the program correctly displaying the Targets list moved along after a turn. **[1 mark]** ▶

```
| | | | | |23|9|140|82|121|34|45|68|75|34|23|119|43|23|119|

Numbers available: 2  3  2  8  512

Current score: 0


Enter an expression: 8+3-2

| | | | |23| |140|82|121|34|45|68|75|34|23|119|43|23|119|119|

Numbers available: 2  3  2  8  512

Current score: 1
```


Task 4 (8 marks)

Coding

- Displaying a suitable menu for the modified standard random game. **[1 mark]**
- Storing the user game type choice in an appropriately named variable. **[1 mark]**
- Storing the "large" numbers in a suitable data structure. **[1 mark]**
- Using selection in the FillNumbers method to fill the NumbersAllowed list differently, depending on the user game type choice. **[1 mark]**
- Correctly populating the NumbersAllowed list with one large number and four standard numbers. **[1 mark]**
- Correctly populating the NumbersAllowed list for the other different game type choices. **[1 mark]**
- Maintaining the functionality to repopulate the NumbersAllowed list in a standard and training game. **[1 mark]**

Teacher Notes:
The difficulty level between turns should be maintained; however, this should be done by repopulating the NumbersAllowed list completely, regardless of whether values within it have been used to identify a target.

Example Solution

Changes to `Main`:

```
            Console.Write("Enter y to play the training game, anything else to play a random game: ");
            string Choice = Console.ReadLine().ToLower();
            Console.WriteLine();
            //CHANGE
            int GameChoiceType = 0;
            if (Choice == "y")
            {
                MaxNumber = 1000;
                MaxTarget = 1000;
                TrainingGame = true;
                Targets = new List<int> { -1, -1, -1, -1, -1, 23, 9, 140, 82, 121, 34, 45, 68, 75, 34, 23, 119, 43, 23, 119 };
            }
            else
            {
                Console.WriteLine("Please select from one of the below game options:");
                Console.WriteLine("1: Standard Game");
                Console.WriteLine("2: Easy Game - One large number and 4 small numbers");
                Console.WriteLine("3: Medium Game - Two large numbers and 3 small numbers");
                Console.WriteLine("4: Hard game - Four large numbers and 1 small number");
                GameChoiceType = Convert.ToInt32(Console.ReadLine());
                MaxNumber = 10;
                MaxTarget = 50;
                TrainingGame = false;
                Targets = CreateTargets(MaxNumberOfTargets, MaxTarget);
            }

            NumbersAllowed = FillNumbers(NumbersAllowed, TrainingGame, MaxNumber, GameChoiceType);
            PlayGame(Targets, NumbersAllowed, TrainingGame, MaxTarget, MaxNumber, GameChoiceType);
            Console.ReadLine();
```


Modification to the `PlayGame` method to expose the game type choice made by the user:

```
        static void PlayGame(List<int> Targets, List<int> NumbersAllowed, bool TrainingGame, int MaxTarget, int MaxNumber, int
GameChoiceType)
        {
            int Score = 0;
            bool GameOver = false;
            string UserInput;
            List<string> UserInputInRPN;
```

…

```
                    if (CheckNumbersUsedAreAllInNumbersAllowed(NumbersAllowed, UserInputInRPN, MaxNumber))
                    {
                        if (CheckIfUserInputEvaluationIsATarget(Targets, UserInputInRPN, ref Score))
                        {
                            RemoveNumbersUsed(UserInput, MaxNumber, NumbersAllowed);
                            NumbersAllowed = FillNumbers(NumbersAllowed, TrainingGame, MaxNumber, GameChoiceType); //CHANGE HERE
                        }
                    }
```

Modification to the `CheckValidNumber` method to recognise a “large” number:

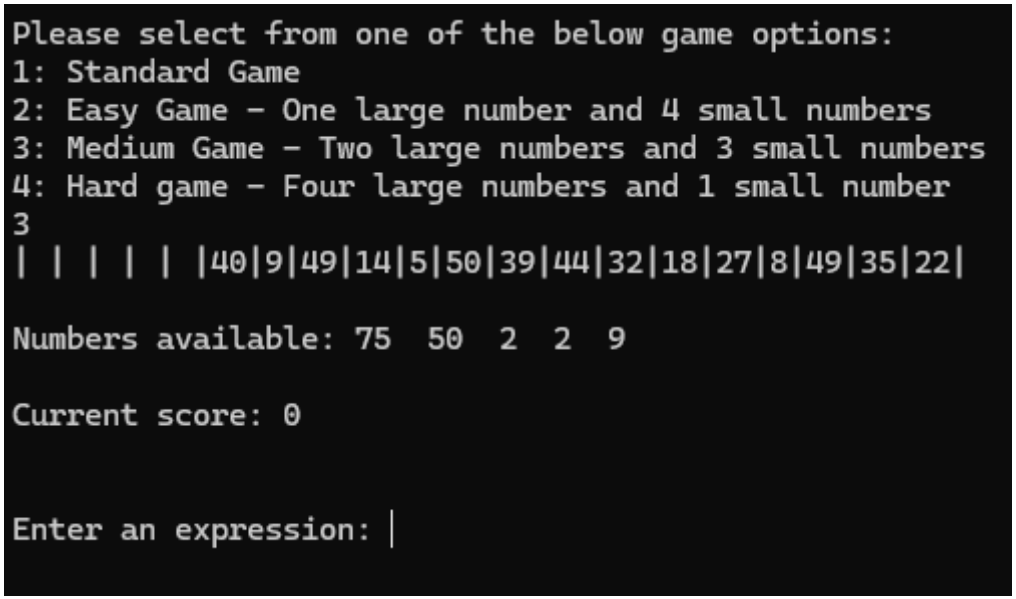
```
        static bool CheckValidNumber(string Item, int MaxNumber)
        {
            if (Regex.IsMatch(Item, "^[0-9]+$"))
            {
                //CHANGE
                int[] Temp = new int[] { 25, 50, 75, 100 };
                int ItemAsInteger = Convert.ToInt32(Item);
                if ((ItemAsInteger > 0 && ItemAsInteger <= MaxNumber) || Temp.Contains(ItemAsInteger))
                {
                    return true;
                }
                //END CHANGE
            }
            return false;
        }
```


Modification to the `FillNumbers` method to correctly fill the `NumbersAllowed` list depending on the game type choice made by the user:

```
        //CHANGE
        static List<int> FillNumbers(List<int> NumbersAllowed, bool TrainingGame, int MaxNumber, int GameChoiceType)
        {
            List<int> LargeNumbers = new List<int>() { 25, 50, 75, 100 };
            if (TrainingGame)
            {
                return new List<int> { 2, 3, 2, 8, 512 };
            }
            else if (GameChoiceType == 2)
            {
                NumbersAllowed = new List<int>();
                NumbersAllowed.Add(LargeNumbers[RGen.Next(4)]);
                for (int i = 0; i < 4; i++)
                {
                    NumbersAllowed.Add(GetNumber(MaxNumber));
                }
            }
            else if (GameChoiceType == 3)
            {
                NumbersAllowed = new List<int>();
                NumbersAllowed.Add(LargeNumbers[RGen.Next(4)]);
                NumbersAllowed.Add(LargeNumbers[RGen.Next(4)]);
                for (int i = 0; i < 3; i++)
                {
                    NumbersAllowed.Add(GetNumber(MaxNumber)); ;
                }
            }
            else if (GameChoiceType == 4)
            {
                NumbersAllowed = new List<int>();
                for (int i = 0; i < 4; i++)
                {
                    NumbersAllowed.Add(LargeNumbers[RGen.Next(4)]);
                }
                NumbersAllowed.Add(GetNumber(MaxNumber));
            }
            else
            {
                while (NumbersAllowed.Count < 5)
                {
                    NumbersAllowed.Add(GetNumber(MaxNumber));
                }
            }
            return NumbersAllowed;
            //END CHANGE
```


Testing

- Displaying two “large” numbers in the NumbersAllowed list. **[1 mark]**



Task 6 (4 marks)

Coding

- Prompting the user to enter an expression or move the targets list. **[1 mark]**
- Iterate through the Targets list moving each item to the right (also accept updating just the head and tail of the list). **[1 mark]**
- Correctly updating the score. **[1 mark]**

Teacher Notes:
The candidate should use selection to ensure the new method MoveTargetsBack is called instead of a normal turn. This is to ensure that MoveTargetsBack and UpdateTargets are not both called together.

The solution shown uses iteration to move the elements of the Targets list one index to the right. This technique is shown because the same technique is used to move the list to the left in the UpdateTargets method. Since the Targets list operates as a queue, the candidate may simply interact with the head and tail of the queue.

Example Solution

Modification to the `PlayGame` method:

```
            while (!GameOver)
            {
                DisplayState(Targets, NumbersAllowed, Score);
                //CHANGE
                Console.Write("Enter an expression or enter MOVE to move the Targets list to the right one step (costs 2 points) : ");
                UserInput = Console.ReadLine();
                Console.WriteLine();
                if (UserInput.ToUpper() == "MOVE")
                {
                    MoveTargetsBack(Targets, ref Score);
                }
                else
                {
```

Creation of new `MoveTargetsBack` method:

```
        //CHANGE
        static void MoveTargetsBack(List<int> Targets, ref int Score)
        {
            for (int Count = Targets.Count - 1; Count > 0; Count--)
            {
                Targets[Count] = Targets[Count - 1];
            }
            Targets[0] = -1;
            Score -= 2;
        }
        //END CHANGE
```


Testing

- Showing the correctly positioned Targets list following the move back. **[1 mark]** ▶

```
Enter an expression or enter MOVE to move the Targets list to the right one step (costs 2 points) : 8+3-2

| | | |23| |140|82|121|34|45| |75|34|23|119|43|23|119|119|119|

Numbers available: 2  3  2  8  512

Current score: 2


Enter an expression or enter MOVE to move the Targets list to the right one step (costs 2 points) : MOVE

| | | | |23| |140|82|121|34|45| |75|34|23|119|43|23|119|119|

Numbers available: 2  3  2  8  512

Current score: 0
```


Task 7 (7 marks)

Coding

- Suitable variable to store a target to be added to the NumbersAllowed list. **[1 mark]**
- Prompt to advise the user of a successful target identified and whether they would like to use it in the NumbersAllowed. **[1 mark]**
- Display the component parts of the identified target for the user to select. **[1 mark]**
- Use of selection to use the correct component part of the identified target. **[1 mark]**
- Add the selected value into the NumbersAllowed list (increasing the list size to six elements). **[1 mark]**
- Selection to maintain the functionality of the FillNumbers method if the user chooses to not use a target number. **[1 mark]**

Teacher Notes:

Candidates should ensure that the FillNumbers method maintains its normal functionality of initially populating the NumbersAllowed list or populating it correctly in the scenario that the user doesn't want to use any numbers from the identified target. The suggested solution uses an optional variable to identify this, which is only assigned a valid number if the user chooses one to use. An alternative technique could use a Boolean variable as a flag to indicate that the FillNumbers method needs to add the additional value.

Example Solution

Changes to the `PlayGame method:`

```
                if (CheckIfUserInputValid(UserInput))
                {
                    UserInputInRPN = ConvertToRPN(UserInput);
                    if (CheckNumbersUsedAreAllInNumbersAllowed(NumbersAllowed, UserInputInRPN, MaxNumber))
                    {
                        //CHANGE
                        int TargetToAddToNumbersAllowed = -1;
                        if (CheckIfUserInputEvaluationIsATarget(Targets, UserInputInRPN, ref TargetToAddToNumbersAllowed, ref Score))
                        {
                            RemoveNumbersUsed(UserInput, MaxNumber, NumbersAllowed);
                            NumbersAllowed = FillNumbers(NumbersAllowed, TrainingGame, MaxNumber, TargetToAddToNumbersAllowed);
                        }
                        //END CHANGE
                    }
                }
```

Changes to the `CheckIfUserInputEvaluationIsATarget` method:

```
        static bool CheckIfUserInputEvaluationIsATarget(List<int> Targets, List<string> UserInputInRPN, ref int
TargetToAddToNumbersAllowed, ref int Score)
        {
            //END CHANGE
            int UserInputEvaluation = EvaluateRPN(UserInputInRPN);
            bool UserInputEvaluationIsATarget = false;
            if (UserInputEvaluation != -1)
            {
                for (int Count = 0; Count < Targets.Count; Count++)
                {
                    if (Targets[Count] == UserInputEvaluation)
```



```
                    {
                        //CHANGE
                        int SuccessfulTarget = Targets[Count];
                        Score += 2;
                        Targets[Count] = -1;
                        UserInputEvaluationIsATarget = true;

                        Console.WriteLine("Target Successfully hit");
                        Console.WriteLine($"Would you like to add either the successful target {SuccessfulTarget} or parts of it to the
Allowed Numbers List? Y/N");
                        string UserChoice = Console.ReadLine().ToUpper();
                        if (UserChoice == "Y")
                        {
                            TargetToAddToNumbersAllowed = SelectValueFromTarget(SuccessfulTarget);
                        }
                        //END CHANGE
                    }
                }
            }
            return UserInputEvaluationIsATarget;
        }
```

Creation of a new `SelectValueFromTarget` method:

```
        //CHANGE
        static int SelectValueFromTarget(int SuccessfulTarget)
        {
            int FirstDigit = SuccessfulTarget / 10;
            int SecondDigit = SuccessfulTarget % 10;

            Console.WriteLine("Please select from the options below which part of the target or the target itself you would like to
include in the Allowed Numbers:");
            Console.WriteLine($"1: The first digit: {FirstDigit}");
            Console.WriteLine($"2: The second digit: {SecondDigit}");
            Console.WriteLine($"3: The whole target: {SuccessfulTarget}");
            int UserChoice = Convert.ToInt32(Console.ReadLine());
            switch (UserChoice)
            {
                case 1:
                    return FirstDigit;
                case 2:
                    return SecondDigit;
                case 3:
                    return SuccessfulTarget;
                default:
                    return -1;
            }
        }
        //END CHANGE
```


Changes to the `FillNumbers` method:

```
        //CHANGE
        static List<int> FillNumbers(List<int> NumbersAllowed, bool TrainingGame, int MaxNumber, int TargetToAddToNumbersAllowed = -1)
        {
            if (TargetToAddToNumbersAllowed == -1)
            {
                if (TrainingGame)
                {
                    return new List<int> { 2, 3, 2, 8, 512 };
                }
                else
                {
                    while (NumbersAllowed.Count < 5)
                    {
                        NumbersAllowed.Add(GetNumber(MaxNumber));
                    }
                    return NumbersAllowed;
                }
            }
            else
            {
                if (TrainingGame)
                {
                    return new List<int> { 2, 3, 2, 8, 512, TargetToAddToNumbersAllowed };
                }
                else
                {
                    while (NumbersAllowed.Count < 5)
                    {
                        NumbersAllowed.Add(GetNumber(MaxNumber));
                    }
                    NumbersAllowed.Add(TargetToAddToNumbersAllowed);
                    return NumbersAllowed;
                }

            }
        }
        //END CHANGE
```


Testing

- Selecting either a correctly identified target or a component part of it and adding it to the NumbersAllowed list. **[1 mark]**

```
Enter an expression: 3*5

Target Successfully hit
Would you like to add either the successful target 15 or parts of it to the Allowed Numbers List? Y/N
y
Please select from the options below which part of the target or the target itself you would like to i
wed Numbers:
1: The first digit: 1
2: The second digit: 5
3: The whole target: 15
2
| | | | |43|22|2|34|32|16|14|25|11|40|3|9| |28|3|21|

Numbers available: 2  3  9  1  4  5

Current score: 1


Enter an expression:
```


Task 10 (11 marks)

Coding

A. Creation of an UndoState class. **[1 mark]**
B. Suitable attributes to store the Targets, NumbersAllowed and Score. **[1 mark]**
C. Constructor which assigns initial values for the Targets, NumbersAllowed and Score. **[1 mark]**
D. Suitable accessor method to expose the class properties. **[1 mark]**
E. Suitable data structure to store undo states which allows LIFO access. **[1 mark]**
F. Advise the user on how many undos are available, if any are. **[1 mark]**
G. Correctly storing a copy of the Score variable in an undo state. **[1 mark]**
H. Correctly storing a copy of the Targets and NumbersAllowed lists in an undo state. **[1 mark]**
I. Correctly restore the Score when undo selected. **[1 mark]**
J. Correctly restore the Targets and NumbersAllowed lists when undo selected. **[1 mark]**

Teacher Notes:
This functionality could be completed using a suitable data structure such as a List or Stack of Tuples containing the Targets list, NumbersAllowed list and Score, rather than a list of objects. Marks B–D can be awarded for suitable list management to access elements in a Stack of Tuples, but full marks should only be awarded if objects are used.

Example Solution

Modification of the `PlayGame` method:

```
        static void PlayGame(List<int> Targets, List<int> NumbersAllowed, bool TrainingGame, int MaxTarget, int MaxNumber)
        {
            //CHANGE
            Stack<UndoState> UndoTurns = new Stack<UndoState>();
            //END CHANGE
            int Score = 0;
            bool GameOver = false;
            string UserInput;
            List<string> UserInputInRPN;
            while (!GameOver)
            {
                DisplayState(Targets, NumbersAllowed, Score);
                //CHANGE
                if (UndoTurns.Count > 0)
                {
                    Console.WriteLine($"There are {UndoTurns.Count} Undos left. Would you like to Undo the last turn? Y/N");
                    string UserChoice = Console.ReadLine().ToUpper();
                    if (UserChoice == "Y")
                    {
                        UndoLastTurn(UndoTurns, NumbersAllowed, Targets, ref Score);
                        DisplayState(Targets, NumbersAllowed, Score);
                    }
                }
                AddtoUndo(UndoTurns, NumbersAllowed, Targets, Score);
                //END CHANGE
```


Creation of new Method UndoLastTurn:

```
        //CHANGE
        static void UndoLastTurn(Stack<UndoState> UndoTurns, List<int> NumbersAllowed, List<int> Targets, ref int Score)
        {
            Targets.Clear();
            NumbersAllowed.Clear();
            UndoState LastMove = UndoTurns.Pop();

            Score = LastMove.GetScore();
            Targets.AddRange(LastMove.GetTargets());
            NumbersAllowed.AddRange(LastMove.GetNumbersAllowed());
        }
```

Creation of new Method AddToUndo:

```
        static void AddtoUndo(Stack<UndoState> UndoTurns, List<int> NumbersAllowed, List<int> Targets, int Score)
        {
            List<int> LastTargetsState = new List<int>();
            List<int> LastNumbersAllowedState = new List<int>();
            foreach (int Item in Targets)
            {
                LastTargetsState.Add(Item);
            }
            foreach (int Item in NumbersAllowed)
            {
                LastNumbersAllowedState.Add(Item);
            }
            UndoTurns.Push(new UndoState(LastTargetsState, LastNumbersAllowedState, Score));

        }
        //END CHANGE
```

Creation of new class UndoState:

```
    //CHANGE
    public class UndoState
    {
        private List<int> Targets;
        private List<int> NumbersAllowed;
        private int Score;

        public UndoState(List<int> targets, List<int> numbersAllowed, int score)
        {
            Targets = targets;
            NumbersAllowed = numbersAllowed;
            Score = score;
```



```
        }
        public List<int> GetTargets()
        {
            return Targets;
        }
        public List<int> GetNumbersAllowed()
        {
            return NumbersAllowed;
        }
        public int GetScore()
        {
            return Score;
        }
    }
    //END CHANGE
```

Testing

- Demonstrating the Targets list and NumbersAllowed list and Score being restored to their state from the previous turn. **[1 mark]**

```
There are 1 Undos left. Would you like to Undo the last turn? Y/N
n
Enter an expression: 8+3-2

| | | |23| |140|82|121|34|45| |75|34|23|119|43|23|119|119|119|

Numbers available: 2  3  2  8  512

Current score: 2


There are 2 Undos left. Would you like to Undo the last turn? Y/N
y
| | | | |23|9|140|82|121|34|45| |75|34|23|119|43|23|119|119|

Numbers available: 2  3  2  8  512

Current score: 1
```


Task 11 (12 marks)

Coding

- Prompt the user if they would like random suggestions for solutions. **[1 mark]**
- Suitable data structure to store solution suggestions without duplication. **[1 mark]**
- Suitable variable to store the number of suggestions attempted to ensure a cap of 30. **[1 mark]**
- Condition-controlled iteration to test solutions. **[1 mark]**
- Suitable variable to store two values from the NumbersAllowed list. **[1 mark]**
- Random selection of mathematical operator (only need +, -, /, *). **[1 mark]**
- Test if the solution found is a target. **[1 mark]**
- Correct testing of expression after division to ensure the evaluation is an integer. **[1 mark]**
- Adding correctly identified targets to storage data structure. **[1 mark]**
- Selection of third operand from NumbersAllowed list and perform mathematical operation with result of first expression evaluation. **[1 mark]**
- Correctly displaying suggested targets * * around them. **[1 mark]**

Teacher Notes:
Candidates should not attempt to find every possible solution using the NumbersAllowed, but instead should simply select two or three operands and operators at random to create an expression within the BIDMAS limitations of the pre-release code. The expression should be evaluated and tested against the Targets list. Once five targets are found, the method should stop and display what it has found. If five possible targets are not identified within 30 attempts, the method should stop looking and display what it has found.

Example Solution

Modification of `PlayGame` method:

```
        static void PlayGame(List<int> Targets, List<int> NumbersAllowed, bool TrainingGame, int MaxTarget, int MaxNumber)
        {
            int Score = 0;
            bool GameOver = false;
            string UserInput;
            List<string> UserInputInRPN;
            while (!GameOver)
            {
                //CHANGE
                Console.WriteLine("Would you like up to five random suggestions of solutions involving only 2 or 3 operands? Y/N");
                string UserChoice = Console.ReadLine().ToUpper();
                HashSet<int> IdentifiedTargets = new HashSet<int>();
                Console.WriteLine();
                if (UserChoice == "Y")
                {
                    IdentifiedTargets = GetRandomSuggestions(NumbersAllowed, Targets);
                    DisplayState(Targets, NumbersAllowed, Score, IdentifiedTargets);
                }
                else
                {
                    DisplayState(Targets, NumbersAllowed, Score);
                }

                //END CHANGE
```


Creation of new `GetRandomSuggestions` method:

```
        //CHANGE
        static HashSet<int> GetRandomSuggestions(List<int> NumbersAllowed, List<int> Targets)
        {
            HashSet<int> Suggestions = new HashSet<int>();
            int SolutionsTested = 0;
            while (Suggestions.Count < 5)
            {
                List<int> Temp = new List<int>();
                foreach (int Item in NumbersAllowed)
                {
                    Temp.Add(Item);
                }
                int FirstOperand = Temp[RGen.Next(Temp.Count)];
                Temp.Remove(FirstOperand);
                int SecondOperand = Temp[RGen.Next(Temp.Count)];
                Temp.Remove(SecondOperand);
                int Operator = RGen.Next(4);
                int Result = 0;
                switch (Operator)
                {
                    case 0:
                        Result = FirstOperand + SecondOperand;
                        if (Targets.Contains(Result) && Result != -1)
                        {
                            Suggestions.Add(Result);
                        }
                        break;
                    case 1:
                        Result = FirstOperand - SecondOperand;
                        if (Targets.Contains(Result) && Result != -1)
                        {
                            Suggestions.Add(Result);
                        }
                        break;
                    case 2:
                        if ((double)FirstOperand / SecondOperand - Math.Truncate((double)FirstOperand / SecondOperand) == 0.0)
                        {
                            Result = (int)FirstOperand / SecondOperand;
                        }
                        if (Targets.Contains(Result) && Result != -1)
                        {
                            Suggestions.Add(Result);
                        }
                        break;
                    case 3:
                        Result = FirstOperand * SecondOperand;
                        if (Targets.Contains(Result) && Result != -1)
                        {
```

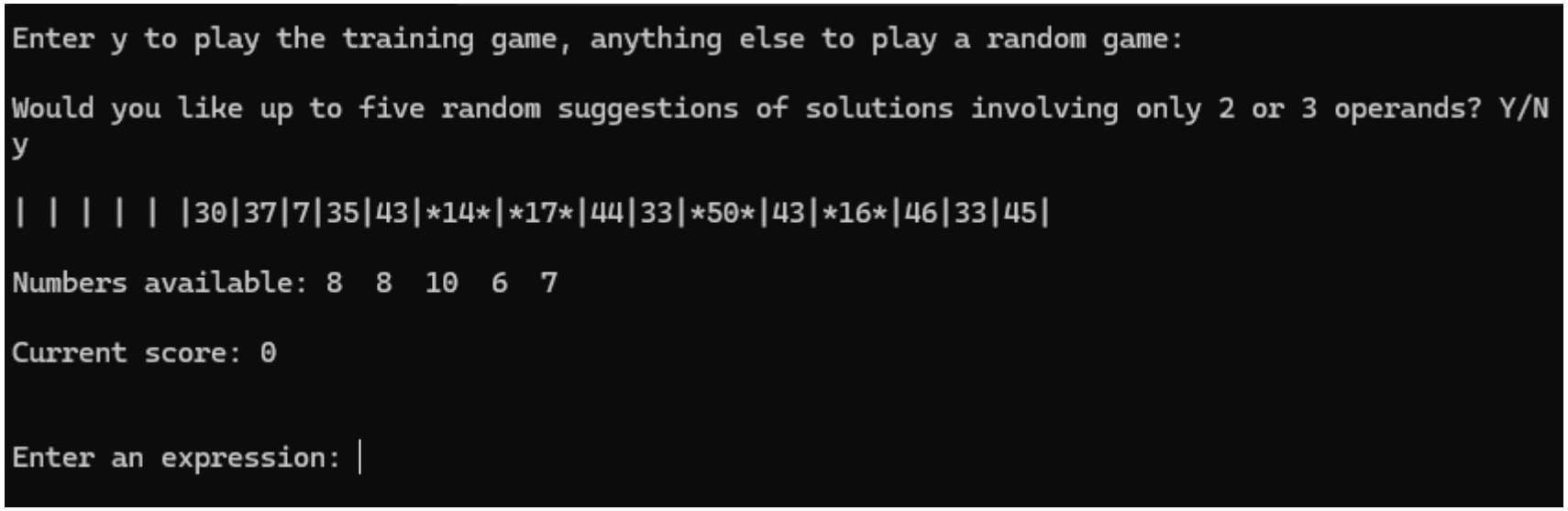


```
                    Suggestions.Add(Result);
                }
                break;
            }
            int ThirdOperand = Temp[RGen.Next(Temp.Count)];
            Operator = RGen.Next(4);
            switch (Operator)
            {
                case 0:
                    Result = Result + ThirdOperand;
                    if (Targets.Contains(Result) && Result != -1)
                    {
                        Suggestions.Add(Result);
                    }
                    break;
                case 1:
                    Result = Result - ThirdOperand;
                    if (Targets.Contains(Result) && Result != -1)
                    {
                        Suggestions.Add(Result);
                    }
                    break;
                case 2:
                    if ((double)Result / ThirdOperand - Math.Truncate((double)Result / ThirdOperand) == 0.0)
                    {
                        Result = (int)Result / ThirdOperand;
                    }
                    if (Targets.Contains(Result) && Result != -1)
                    {
                        Suggestions.Add(Result);
                    }
                    break;
                case 3:
                    Result = Result * ThirdOperand;
                    if (Targets.Contains(Result))
                    {
                        Suggestions.Add(Result);
                    }
                    break;
            }

            SolutionsTested++;
            if (SolutionsTested > 30)
            {
                break;
            }
        }
        return Suggestions;
    }
    //END CHANGE
```


Testing

- Displaying correct suggestion targets in the Targets list. **[1 mark]**



Task 12 (7 marks)

Coding

- Suitable variable to store the evaluation of an expression entered by the user. **[1 mark]**
- Suitable variable to store the number of targets which the evaluation matches. **[1 mark]**
- Advising the user of the evaluation of their expression and the number of target matches. **[1 mark]**
- Suitable data structure to store invalid numbers in a user expression. **[1 mark]**
- Advising the user which numbers in their expression are invalid (if required). **[1 mark]**

Example Solution

Modification to the `CheckIfUserInputEvaluationIsATarget` method:

```
        static bool CheckIfUserInputEvaluationIsATarget(List<int> Targets, List<string> UserInputInRPN, ref int Score)
        {
            //CHANGE
            int TargetsFound = 0;
            int UserInputEvaluation = EvaluateRPN(UserInputInRPN);
            bool UserInputEvaluationIsATarget = false;
            if (UserInputEvaluation != -1)
            {
                for (int Count = 0; Count < Targets.Count; Count++)
                {
                    if (Targets[Count] == UserInputEvaluation)
                    {
                        Score += 2;
                        Targets[Count] = -1;
                        UserInputEvaluationIsATarget = true;
                        TargetsFound++;
                    }
                }
            }
            Console.WriteLine($"Your expression evaluated to {UserInputEvaluation} and was found {TargetsFound} times.");
            Console.WriteLine();
            return UserInputEvaluationIsATarget;
        }
        //END CHANGE
```


Creation of new CheckNumbersUsedAreAllInNumbersAllowed method:

```
        static bool CheckNumbersUsedAreAllInNumbersAllowed(List<int> NumbersAllowed, List<string> UserInputInRPN, int MaxNumber)
        {
            //CHANGE
            bool AllNumbersValid = true;
            List<int> InvalidNumbers = new List<int>();
            List<int> Temp = new List<int>();
            foreach (int Item in NumbersAllowed)
            {
                Temp.Add(Item);
            }
            foreach (string Item in UserInputInRPN)
            {
                if (CheckValidNumber(Item, MaxNumber))
                {
                    if (Temp.Contains(Convert.ToInt32(Item)))
                    {
                        Temp.Remove(Convert.ToInt32(Item));
                    }
                    else
                    {
                        InvalidNumbers.Add(Convert.ToInt32(Item));
                        AllNumbersValid = false;
                    }
                }
            }
            if (!AllNumbersValid)
            {
                Console.WriteLine("Your expression is not valid because the following numbers are not available to you:");
                foreach (int Item in InvalidNumbers)
                {
                    Console.Write($"{Item}, ");
                }
                Console.WriteLine();
                Console.WriteLine();
            }
            return AllNumbersValid;
            //END CHANGE
        }
```


Testing

- Advising the user that they have entered an invalid number in their expression. **[1 mark]**

```
Enter an expression: 8+100

Your expression is not valid because the following numbers are not available to you:
100,

| | | | |23|9|140|82|121|34|45|68|75|34|23|119|43|23|119|119|

Numbers available: 2  3  2  8  512

Current score: -1


Enter an expression:
```

- Advising the user that they have found a valid target and how many times it appears in the Targets list. **[1 mark]**

```
Enter an expression: 8+3-2

Your expression evaluated to 9 and was found 1 times.

| | | |23| |140|82|121|34|45|68|75|34|23|119|43|23|119|119|119|

Numbers available: 2  3  2  8  512

Current score: 0
```


Task 13 (7 marks)

Coding

- Use of MaxNumberOfTargets variable as upperbound when repopulating the Targets list. **[1 mark]**
- Suitable counting variable to count the number of targets to be removed / calculate the size of slice. **[1 mark]**
- Suitable technique for identifying second target. **[1 mark]**
- Removing slice / range from Targets list. **[1 mark]**
- Targets list repopulated to the correct length. **[1 mark]**
- Targets list repopulated with correct values for both training game and standard random game. **[1 mark]**

Teacher Notes:
Removal of targets from the Targets list could also be achieved through iteration.

Example Solution

Modification to the `Main` method:

```
            }
            NumbersAllowed = FillNumbers(NumbersAllowed, TrainingGame, MaxNumber);
            //CHANGE
            PlayGame(Targets, NumbersAllowed, TrainingGame, MaxTarget, MaxNumber, MaxNumberOfTargets);
            //END CHANGE
            Console.ReadLine();
        }
```

Modification of the `PlayGame` method:

```
        //CHANGE
        static void PlayGame(List<int> Targets, List<int> NumbersAllowed, bool TrainingGame, int MaxTarget, int MaxNumber, int
MaxNumberOfTargets)
        {
            //END CHANGE
            int Score = 0;
```

…

```
                else
                {
                    //CHANGE
                    UpdateTargets(Targets, TrainingGame, MaxTarget, MaxNumberOfTargets);
                    //END CHANGE
                }
            }
            Console.WriteLine("Game over!");
            DisplayScore(Score);
        }
```


Modification of the `CheckIfUserInputEvaluationIsATarget` method:

```
            if (UserInputEvaluation != -1)
            {
                for (int Count = 0; Count < Targets.Count; Count++)
                {
                    if (Targets[Count] == UserInputEvaluation)
                    {
                        Score += 2;
                        //CHANGE
                        int TargetsToRemove = 0;
                        for (int i = Count + 1; i < Targets.Count; i++)
                        {
                            TargetsToRemove++;
                            if (Targets[i] == UserInputEvaluation)
                            {
                                Score += TargetsToRemove * 2;
                                Targets.RemoveRange(Count + 1, TargetsToRemove);  //Removes all the elements between the first and the
last found including the last found
                                break;
                            }
                        }
                        Targets.RemoveAt(Count);    //Remove the first one
                        UserInputEvaluationIsATarget = true;
                        //END CHANGE
                    }
                }
            }
```

Modification of the `UpdateTargets` method:

```
        //CHANGE
        static void UpdateTargets(List<int> Targets, bool TrainingGame, int MaxTarget, int MaxNumberOfTargets)
        {
            for (int Count = 0; Count < Targets.Count - 1; Count++)
            {
                Targets[Count] = Targets[Count + 1];
            }
            Targets.RemoveAt(Targets.Count - 1);
            if (TrainingGame)
            {
                while (Targets.Count < MaxNumberOfTargets)
                {
                    Targets.Add(Targets[Targets.Count - 1]);
                }

            }
            else
```



```
        {
            while (Targets.Count < MaxNumberOfTargets)
            {
                Targets.Add(GetTarget(MaxTarget));
            }
        }
    }
    //END CHANGE
```

Testing

- Show all the values between the two 34 targets (inclusive) being removed, Targets list being backfilled with 119s and the score increasing to 9 points. **[1 mark]**

```
Enter y to play the training game, anything else to play a random game: y

| | | | | |23|9|140|82|121|34|45|68|75|34|23|119|43|23|119|

Numbers available: 2  3  2  8  512

Current score: 0


Enter an expression: 512/8/2+2

| | | | |23|9|140|82|121|23|119|43|23|119|119|119|119|119|119|119|

Numbers available: 2  3  2  8  512

Current score: 9


Enter an expression:
```


Task 15 (10 marks)

Coding

- Suitable adjustment of scope of the Score variable to allow it to be updated in a previous game state load. **[1 mark]**
- Prompt inviting the user to load a previous game state. **[1 mark]**
- Correct assignment of the TrainingGame, MaxNumber and MaxTarget variables in all scenarios. **[1 mark]**
- Condition-controlled loop while invalid code is entered by the user. **[1 mark]**
- Correct checking that the code only contains valid characters. *This could be done with string handling, ASCII conversion or Regular Expression*. **[1 mark]**
- Correct checking that the restore code is the right length. **[1 mark]**
- Correct checking that NumbersAllowed and Score portions of the code do not contain the @ symbol. **[1 mark]**
- Correct restoration of the Targets list from a previous game state code. **[1 mark]**
- Correct restoration of the NumbersAllowed list and Score from a previous game state code. **[1 mark]**

Example Solution

Modification of the `Main` method:

```
            //CHANGE
            int Score = 0;
            Console.Write("Enter y to play the training game, L to restore a previous game state, anything else to play a random game:
");
            string Choice = Console.ReadLine().ToUpper();
            Console.WriteLine();
            if (Choice == "y")
            {
                MaxNumber = 1000;
                MaxTarget = 1000;
                TrainingGame = true;
                Targets = new List<int> { -1, -1, -1, -1, -1, 23, 9, 140, 82, 121, 34, 45, 68, 75, 34, 23, 119, 43, 23, 119 };
                NumbersAllowed = FillNumbers(NumbersAllowed, TrainingGame, MaxNumber);
            }
            else if (Choice == "L")
            {
                RestoreGame(Targets, NumbersAllowed, ref Score);
                TrainingGame = false;
                MaxNumber = 10;
                MaxTarget = 50;
            }
            else
            {
                MaxNumber = 10;
                MaxTarget = 50;
                TrainingGame = false;
                Targets = CreateTargets(MaxNumberOfTargets, MaxTarget);
                NumbersAllowed = FillNumbers(NumbersAllowed, TrainingGame, MaxNumber);
            }
            PlayGame(Targets, NumbersAllowed, TrainingGame, MaxTarget, MaxNumber, ref Score);
            Console.ReadLine();
            //END CHANGE
```


Creation of the RestoreGame method:

```
        //CHANGE
        static void RestoreGame(List<int> Targets, List<int> NumbersAllowed, ref int Score)
        {
            string RestoreCode = "";
            bool ValidCode = false;
            while (!ValidCode)
            {
                Console.WriteLine("Enter in the game restore code. It should be 26 characters.");
                RestoreCode = Console.ReadLine();
                if (Regex.IsMatch(RestoreCode, "^[@A-z[\\]^_']+$"))
                {
                    if (RestoreCode.Length == 26)
                    {
                        if (RestoreCode.Substring(20, 6).Contains("@"))     //There cannot be any missing chars in NumbersAllowed or
the score
                        {
                            Console.WriteLine("That is not a valid code");
                        }
                        else
                        {
                            ValidCode = true;
                        }
                    }
                    else
                    {
                        Console.WriteLine("That code is not the right length");
                    }
                }
                else
                {
                    Console.WriteLine("That code contains invalid characters");
                }
            }
            for (int i = 0; i < 20; i++)
            {
                if ((Convert.ToInt32(RestoreCode[i]) - 64) == 0)
                {
                    Targets.Add(-1);
                }
                else
                {
                    Targets.Add(Convert.ToInt32(RestoreCode[i]) - 64);
                }
```



```
            }
            for (int i = 20; i < 25; i++)
            {
                NumbersAllowed.Add(Convert.ToInt32(RestoreCode[i]) - 64);
            }
            Score = Convert.ToInt32(RestoreCode[25]) - 64;
            Console.WriteLine();

        }
        //END CHANGE
```

Modification of the `PlayGame` method:

```
        //CHANGE
        static void PlayGame(List<int> Targets, List<int> NumbersAllowed, bool TrainingGame, int MaxTarget, int MaxNumber, ref int Score)
        {
            //END CHANGE
            //CHANGE - removed Score from this scope and passed in a parameter

            bool GameOver = false;
            string UserInput;
```

Testing

- Displaying game restore code and correctly restored game. **[1 mark]**

```
Enter y to play the training game, L to restore a previous game state, anything else to play a random game: L

Enter in the game restore code. It should be 26 characters.
@@@@@Qf_Hg@kCCN@WJhSCHGJAC

| | | | | |17|38|31|8|39| |43|3|3|14| |23|10|40|19|

Numbers available: 3  8  7  10  1

Current score: 3


Enter an expression:
```


Task 16 (12 marks)

Coding

- Use of a Stack data structure to store operators. **[1 mark]**
- Use of a Queue data structure to store the postfix notation. **[1 mark]**
- Conversion of the infix expression into individual elements. *This could also be done using iteration.* **[1 mark]**
- Data structure to suitably handle precedence to correctly handle open parenthesis. **[1 mark]**
- Correctly pushing operators onto a stack in the right order according to precedence. **[1 mark]**
- Correctly recognising the start and end brackets in an expression. **[1 mark]**
- Correctly popping items from the stack to remove all items inside brackets and enqueue them in the right order. **[1 mark]**
- Correctly removing operators from the stack without underflowing. **[1 mark]**
- Correctly enqueuing operands into a queue data structure. **[1 mark]**
- Correctly matching the parenthesis symbols in the CheckValidOperator method. **[1 mark]**
- Correct modification of the Regular Expression to match an infix expression with brackets. **[1 mark]**

Teacher Notes:
Using only the Regular Expression meta characters from the AQA specification, it is not possible to count the number of brackets used in an infix expression to ensure that it is mathematically correct. Whilst it could be done with backtracking, that is not a requirement for this question and candidates should not be credited for it. The candidates can assume a mathematically valid input.

The suggested solution uses a new method ConvertToRPNWithBrackets rather than modifying the existing ConvertToRPN method because the number of changes which were needed would mean the original method would need significant modifications.

Example Solution

Replacement of the `ConvertToRPN` method with the new `ConvertToRPNWithBrackets` method wherever it is called:

```
##PlayGame
            if (CheckIfUserInputValid(UserInput))
              {
                  //CHANGE
                  UserInputInRPN = ConvertToRPNWithBrackets(UserInput, MaxNumber);
                  //END CHANGE
                  if (CheckNumbersUsedAreAllInNumbersAllowed(NumbersAllowed, UserInputInRPN, MaxNumber))
                  {
                      if (CheckIfUserInputEvaluationIsATarget(Targets, UserInputInRPN, ref Score))
                      {
                          RemoveNumbersUsed(UserInput, MaxNumber, NumbersAllowed);
                          NumbersAllowed = FillNumbers(NumbersAllowed, TrainingGame, MaxNumber);
                      }
                  }
              }
```

…


```
##RemoveNumbersUsed
        static void RemoveNumbersUsed(string UserInput, int MaxNumber, List<int> NumbersAllowed)
        {
            //CHANGE
            List<string> UserInputInRPN = ConvertToRPNWithBrackets(UserInput, MaxNumber);
            //END CHANGE
            foreach (string Item in UserInputInRPN)
            {
                if (CheckValidNumber(Item, MaxNumber))
                {
                    if (NumbersAllowed.Contains(Convert.ToInt32(Item)))
                    {
                        NumbersAllowed.Remove(Convert.ToInt32(Item));
                    }
                }
            }
        }
```

Creation of new ConvertToRPNWithBrackets method:

```
        //CHANGE
        static List<string> ConvertToRPNWithBrackets(string UserInput, int MaxNumber)
        {
            Stack<string> Operators = new Stack<string>();
            Queue<string> RPNQueue = new Queue<string>();

            string[] FullInfixExpression = Regex.Split(UserInput, @"(\D)");
            List<string> CleanedInfixExpression = new List<string>();

            foreach (string Item in FullInfixExpression)
            {
                if (!string.IsNullOrEmpty(Item))
                {
                    CleanedInfixExpression.Add(Item);
                }
            }
            Dictionary<string, int> Precedence = new Dictionary<string, int>
            {
                {"(", 1 }, { "+", 2 }, { "-", 2 }, { "*", 4 }, { "/", 4 }
            };
            foreach (string Item in CleanedInfixExpression)
            {
                if (CheckValidOperator(Item))
                {
                    if (Item == "(")
                    {
                        Operators.Push(Item);
```



```
                }
                else if (Item == ")")
                {
                    while (Operators.Count > 0)
                    {
                        if (Operators.Peek() != "(")
                        {
                            RPNQueue.Enqueue(Operators.Pop());
                        }
                        else
                        {
                            Operators.Pop();
                            break;
                        }
                    }
                }
                else if (Operators.Count > 0 && Precedence[Operators.Peek()] > Precedence[Item])
                {
                    RPNQueue.Enqueue(Operators.Pop());
                    Operators.Push(Item);
                }
                else
                {
                    Operators.Push(Item);
                }
            }
            else if (CheckValidNumber(Convert.ToString(Item), MaxNumber))
            {
                RPNQueue.Enqueue(Convert.ToString(Item));
            }
        }
        while (Operators.Count > 0)
        {
            RPNQueue.Enqueue(Operators.Pop());
        }
        return RPNQueue.ToList();
    }
    //END CHANGE
```

Modification of the Regular Expression in the `CheckValidOperator` method:

```
    //CHANGE
    static bool CheckValidOperator(string Item)
    {
        return Regex.IsMatch(Item, @"[\+\-\*\/\(\)]");
    }
    //END CHANGE
```


Modification of the Regular Expression in the `CheckIfUserInputValid` method:

```
        //CHANGE
        static bool CheckIfUserInputValid(string UserInput)
        {
            return Regex.IsMatch(UserInput, @"^\(*([0-9]+|[\+\-\*\/\(\)])+\)*$");
        }
        //END CHANGE
```

Testing

- Showing a target correctly identified using an infix expression including brackets. **[1 mark]**

```
Enter y to play the training game, anything else to play a random game: y

| | | | | |23|9|140|82|121|34|45|68|75|34|23|119|43|23|119|

Numbers available: 2  3  2  8  512

Current score: 0


Enter an expression: (8+2)*2+3

| | | | | |9|140|82|121|34|45|68|75|34| |119|43| |119|119|

Numbers available: 2  3  2  8  512

Current score: 5


Enter an expression:
```


Task 19 (14 marks)

Coding

- Prompt to ask the user if they would like helper suggestions. **[1 mark]**
- Selection to branch program appropriately depending on their choice to display helper suggestions. **[1 mark]**
- Suitable data structure to store text expressions and associated evaluations. **[1 mark]**
- Count-controlled loop to iterate through data structure storing text expressions and associated evaluations. **[1 mark]**
- Iterating through the NumbersAllowed list to test permutations. **[1 mark]**
- Rotating the NumbersAllowed list (or similar functionality) to test different permutations of numbers. **[1 mark]**
- Appropriately displaying the combination of text expressions and associated evaluations on the screen for the user. **[1 mark]**
- Use of recursion to try combinations. **[1 mark]**
- Only storing suggestion solutions for targets which have not already been identified. **[1 mark]**
- Correctly calculating expressions which use division to ensure they evaluate to an integer. **[1 mark]**
- Testing expressions to ensure they correctly follow BIDMAS if needed (required for expressions built up through recursion). **[1 mark]**
- Generate expressions which can use the four mathematical operators: + - / * **[1 mark]**
- Storage of expression with associated evaluation. **[1 mark]**

Teacher Notes:
This functionality could be completed using iteration. Marks should be awarded for techniques, but full marks should only be awarded if recursion is used.

Because the expression is built up step by step, it must be tested at each stage because the impact of BIDMAS may change the evaluation as the expression builds.

Example Solution

Modification of the `PlayGame` method:

```
            while (!GameOver)
            {
                DisplayState(Targets, NumbersAllowed, Score);
                //CHANGE
                Console.WriteLine("Would you like helper suggestions: Y/N");
                string UserChoice = Console.ReadLine().ToUpper();
                if (UserChoice == "Y")
                {
                    List<int> Temp = new List<int>();
                    Dictionary<int, string> PossibleSolutions = new Dictionary<int, string>();
                    foreach (int Item in NumbersAllowed)
                    {
                        Temp.Add(Item);
                    }
                    for (int i = 0; i < 5; i++)
                    {
                        Dictionary<int, string> TestSolutions = GenerateEvaluations(Temp, Targets);
                        foreach (KeyValuePair<int, string> Solution in TestSolutions)
                        {
```



```
                        if (!PossibleSolutions.ContainsKey(Solution.Key))
                        {
                            PossibleSolutions.Add(Solution.Key, Solution.Value);
                        }
                    }
                    Temp.Add(Temp[0]);
                    Temp.RemoveAt(0);
                }
                Console.WriteLine();
                foreach (KeyValuePair<int, string> Solution in PossibleSolutions)
                {
                    Console.WriteLine($"{Solution.Key} can be calculated using the expression: {Solution.Value}");
                }
                Console.WriteLine();
            }
            //END CHANGE
            Console.Write("Enter an expression: ");
            UserInput = Console.ReadLine();
```

Creation of new `GenerateEvaluations` method (and associated helper method):

```
        //CHANGE
        public static Dictionary<int, string> GenerateEvaluations(List<int> NumbersAllowed, List<int> Targets)
        {
            Dictionary<int, string> PossibleExpressions = new Dictionary<int, string>();
            GenerateEvaluationsHelper(NumbersAllowed, Targets, 0, NumbersAllowed[0], PossibleExpressions,
NumbersAllowed[0].ToString());
            return PossibleExpressions;
        }

        private static void GenerateEvaluationsHelper(List<int> NumbersAllowed, List<int> Targets, int Index, int CurrentResult,
Dictionary<int, string> PossibleExpressions, string CurrentExpression)
        {
            if (Index == NumbersAllowed.Count - 1)
            {
                //Because the recursion calculates expressions step by step rather than as one whole expression
                //the new code needs to test the end result using RPN evaluator to ensure BIDMAS is correctly followed
                if (Targets.Contains(EvaluateRPN(ConvertToRPN(CurrentExpression))) && EvaluateRPN(ConvertToRPN(CurrentExpression)) != -
1)
                {
                    if (!PossibleExpressions.ContainsKey(EvaluateRPN(ConvertToRPN(CurrentExpression))))
                    {
                        PossibleExpressions.Add(EvaluateRPN(ConvertToRPN(CurrentExpression)), CurrentExpression);
                    }
                }
                return;
            }
```



```
            int NextNumber = NumbersAllowed[Index + 1];
            GenerateEvaluationsHelper(NumbersAllowed, Targets, Index + 1, CurrentResult * NextNumber, PossibleExpressions,
$"{CurrentExpression}*{NextNumber}");
            if (NextNumber != 0)
            {
                if ((double)CurrentResult / NextNumber - Math.Truncate((double)CurrentResult / NextNumber) == 0.0)
                {
                    GenerateEvaluationsHelper(NumbersAllowed, Targets, Index + 1, (int)CurrentResult / NextNumber, PossibleExpressions,
$"{CurrentExpression}/{NextNumber}");
                }
            }
            GenerateEvaluationsHelper(NumbersAllowed, Targets, Index + 1, CurrentResult + NextNumber, PossibleExpressions,
$"{CurrentExpression}+{NextNumber}");
            GenerateEvaluationsHelper(NumbersAllowed, Targets, Index + 1, CurrentResult - NextNumber, PossibleExpressions,
$"{CurrentExpression}-{NextNumber}");
        }
        //END CHANGE
```

Testing

- Show the program displaying the suggested valid expressions for targets. **[1 mark]**