**Sample Practice for Pre-Release (Task 5 / 7 / 10)**

**Task 5** **Marks:** 4

This question is about the use of the values in the NumbersAllowed list. The application currently awards the same number of points regardless of the complexity of the expression entered by the user. This does not encourage mathematical extension and challenge.

Introduce functionality into the application to give a bonus of 2 points for every operand used in the expression entered, if the evaluation is a valid target.

**What you need to do**

**Task 5.1**

Modify the CheckIfUserInputEvaluationIsATarget method to operate as described.

**Task 5.2**

Test that the changes you have made work:

* Run the Skeleton Program.
* Enter y to start a training game.
* Enter the expression: 512/8+2+2
* Show the program correctly identifying the target 68 and increasing the user score to 9 points (8 additional points for four operands used in the expression).

**Evidence that you need to provide:**

* Your PROGRAM SOURCE CODE showing the amended CheckIfUserInputEvaluationIsATarget method. [3 marks]
* SCREEN CAPTURE(S) showing the required tests. [1 mark]

**Mr Chambers HINTS**

* **Do you have suitable variable names that will help you count the number of operands to it?**
* **Can you iterate through the users expression, for loop with the length of targets might help for this.**
* **Did you know that you can use isdigit to check for numbers. Just be aware that it doesn’t pick up negative numbers but could be useful for this program.**
* **The bonus score should probably be added before the score is decremented between turns so that you can maintain the normal turn to turn operations in the program.**

|  |
| --- |
| **[Program / Changes here]**  def CheckIfUserInputEvaluationIsATarget(Targets, UserInputInRPN, Score):  #edits start here  count = 0  for x in UserInputInRPN:  count += 1 if x.isdigit() else 0  #edits end here  UserInputEvaluation = EvaluateRPN(UserInputInRPN)  UserInputEvaluationIsATarget = False  if UserInputEvaluation != -1:  for Count in range(0, len(Targets)):  if Targets[Count] == UserInputEvaluation:  Score += 2  Targets[Count] = -1  UserInputEvaluationIsATarget = True  #edit to increase score  Score += count \* 2  #edit end  return UserInputEvaluationIsATarget, Score |
| **[Testing here]** |

**Correct – 4 marksCor**

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

This question extends the Skeleton Program to allow the user to use correctly identified targets in   
future expressions.

For example, in a standard random game with a Targets list and a NumbersAllowed list of:

| | | | | |43|46|24|42|3|15|27|35|8|28|5|4|3|35|36|

Numbers available: 1 9 6 8 2

If the user enters the expression 9+6, this correctly identifies the target number 15. The user should then be given the option to either add the number 15 or just 1 or just 5 into the NumbersAllowed list to be used in a later turn. The NumbersAllowed list can increase to more than five values. This functionality can add a 0 to the NumbersAllowed list if the identified target is a single digit or is divisible by 10.

Introduce new functionality, so that when a target is correctly identified, give the user the option to add the target or part of it into their NumbersAllowed list. The program should display the target and its component digits to the user to allow them to choose. If they select to add the target (or one of its component digits) it should be added to the NumbersAllowed list.

**What you need to do**

**Task 7.1**

Modify the PlayGame and CheckIfUserInputEvaluationIsATarget methods to display when a target has been correctly identified and prompt the user to choose if they would like to use it to add into the NumbersAllowed list.

Create a new method called SelectValueFromTarget which displays the correctly identified target and its component digits and invites the user to select which part they would like to add to the NumbersAllowed list.

**Task 7.2**

Modify the FillNumbers method to add the selected target from the user to the NumbersAllowed list whilst maintaining its functionality at the start of the game of initially populating the NumbersAllowed list.

**Task 7.3**

Test that the changes you have made work:

* Run the Skeleton Program.
* Press enter to start a standard random game.
* Enter an expression which correctly identifies a target greater than 9.
* Show the program prompting the user to choose if they would like to use the target in their NumbersAllowed list. Select this option.
* Show the program prompting the user which digit in the target they would like to use. The program should show the target and both digits individually. Select one of the options available.
* Show the program correctly adding the selected option to the NumbersAllowed list.

**Evidence that you need to provide:**

* Your PROGRAM SOURCE CODE showing the amended CheckIfUserInputEvaluationIsATarget and PlayGame methods. [2 marks]
* Your PROGRAM SOURCE CODE showing the new SelectValueFromTarget method. [2 marks]
* Your PROGRAM SOURCE CODE showing the amended FillNumbers method. [2 marks]
* SCREEN CAPTURE(S) showing the required tests. [1 mark]

**Mr Chambers hints / tips for question 7: -**

* **Have you got a suitable variable to store a target and is it added to the numbersallowed list.**
* **Is there a prompt to ask the user about the successful target identified and whether they would like to use it in the numbers allowed.**
* **Hint… Targettoaddtonumbersallowed = -1**
* **You can add multiple parameters i.e TargetToAdd, IsTarget, Score = (function(4 parameters go here) …**
* **Fill numbers method will need to have selection if the user decides not to use a target number.**

|  |
| --- |
| **[Play game changes here]**  def PlayGame(Targets, NumbersAllowed, TrainingGame, MaxTarget, MaxNumber):  Score = 0  GameOver = False  while not GameOver:  DisplayState(Targets, NumbersAllowed, Score)  UserInput = input("Enter an expression: ")  print()  if CheckIfUserInputValid(UserInput):  UserInputInRPN = ConvertToRPN(UserInput)  if CheckNumbersUsedAreAllInNumbersAllowed(NumbersAllowed, UserInputInRPN, MaxNumber):  #edits to code  IsTarget, Score, TargetVal = CheckIfUserInputEvaluationIsATarget(Targets, UserInputInRPN, Score)  if IsTarget:  print(f"\nTarget {TargetVal} has been correctly identified!")  add\_choice = input("Would you like to add this target or part of it to your NumbersAllowed list? (y/n): ").lower()  targettonumbersallowed = -1  if add\_choice == "y":  targettonumbersallowed = SelectValueFromTarget(TargetVal)  NumbersAllowed = RemoveNumbersUsed(UserInput, MaxNumber, NumbersAllowed)  NumbersAllowed = FillNumbers(NumbersAllowed, TrainingGame, MaxNumber, targettonumbersallowed, TargetVal)  #end of edits  else:  Score -= 1  else:  Score -= 1  else:  Score -= 1  if Targets[0] != -1:  GameOver = True  else:  Targets = UpdateTargets(Targets, TrainingGame, MaxTarget)  print("Game over!")  DisplayScore(Score) |
| **[Checkifuserinputevaluationisatarget changes here]**  def CheckIfUserInputEvaluationIsATarget(Targets, UserInputInRPN, Score):  UserInputEvaluation = EvaluateRPN(UserInputInRPN)  UserInputEvaluationIsATarget = False  #edits start  TargetIdentified = -1  if UserInputEvaluation != -1:  for Count in range(0, len(Targets)):  if Targets[Count] == UserInputEvaluation:  Score += 2  Targets[Count] = -1  UserInputEvaluationIsATarget = True  TargetIdentified = UserInputEvaluation  break  return UserInputEvaluationIsATarget, Score, TargetIdentified  #end of edits |
| **[Creation of a new select value from target method]**  #new function  def SelectValueFromTarget(TargetVal):  print(f"Target evaluated: {TargetVal}")  digits = list(str(TargetVal))  print("Component digits: " + ", ".join(digits))  print("Enter the target itself or one of its digits to add to your list.")  choice = input(f"Choose from {[TargetVal] + digits}")  if not choice.isdigit():  return -1  targettonumbersallowed = int(choice)  return targettonumbersallowed  #end of edits |
| **[Changes to fill numbers method here]**  #edits to code  def FillNumbers(NumbersAllowed, TrainingGame, MaxNumber, targettonumbersallowed=None, target\_val=None):  if targettonumbersallowed is not None and targettonumbersallowed != -1:  NumbersAllowed.append(targettonumbersallowed)  if target\_val is not None and ((target\_val < 10) or (target\_val % 10 == 0)):  NumbersAllowed.append(0)  #end of edits  if TrainingGame and len(NumbersAllowed) < 5:  return [2, 3, 2, 8, 512]  else:  while len(NumbersAllowed) < 5:  NumbersAllowed.append(GetNumber(MaxNumber))  return NumbersAllowed |
| **[print screen of testing]** |

**6/7 marks – Needed to make the numbers allowed list a length of 6 if user chooses to add the target to NumbersAllowed**

**[Changes to fill numbers method here]**

#edits to code

def FillNumbers(NumbersAllowed, TrainingGame, MaxNumber, targettonumbersallowed=None, target\_val=None):

if targettonumbersallowed is not None and targettonumbersallowed != -1:

**NumbersAllowed.append(GetNumber(MaxNumber))**

NumbersAllowed.append(targettonumbersallowed)

if target\_val is not None and ((target\_val < 10) or (target\_val % 10 == 0)):

NumbersAllowed.append(0)

#end of edits

if TrainingGame and len(NumbersAllowed) < 5:

return [2, 3, 2, 8, 512]

else:

while len(NumbersAllowed) < 5:

NumbersAllowed.append(GetNumber(MaxNumber))

return NumbersAllowed

**Task 10** **Marks:** 11

This question extends the Skeleton Program by introducing some object orientation to allow the user to undo previous moves.

Introduce new functionality to allow the user to undo their previous moves. If there are undo moves available, the program should inform the user how many are available and ask them if they would like to undo their last move. If they select this option, the program should undo the last move.

**What you need to do**

**Task 10.1**

Create a new class called UndoState which stores the copies of the NumbersAllowed list, Targets list and Score. Create suitable accessor methods to be able to access these properties.

Modify the PlayGame method to prompt the user appropriately and introduce a suitable data structure to store UndoState objects to fulfil the requirements described.

**Task 10.2**

Create a new method AddToUndo which instantiates a copy of the current game state and stores it accordingly to allow undo functionality.

Create a new method UndoLastTurn which restores the game to the previous game state, ensuring that the NumbersAllowed list, Targets list and Score are all updated.

**Task 10.3**

Test that the changes you have made work:

* Run the Skeleton Program.
* Enter y to start a training game.
* Enter the expression: 512/8+2+2
* When prompted, do not undo the last turn.
* Enter the expression: 8+3-2
* Show the program displaying that 2 undos are available.
* When prompted undo the last move.
* Show the program displaying the Targets list:

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

**Evidence that you need to provide:**

* Your PROGRAM SOURCE CODE showing the new class UndoState with a suitable   
  constructor and accessor methods. [4 marks]
* Your PROGRAM SOURCE CODE showing the amended PlayGame method with a   
  suitable prompt as described. [2 marks]
* Your PROGRAM SOURCE CODE showing the new AddToUndo and UndoLastTurn   
  methods. [4 marks]
* SCREEN CAPTURE(S) showing the required tests. [1 mark]

**Mr Chambers Hints / Tips:**

* You will need to create a new class called ‘undostate’
* You should have 3 parameters passed in called target, numbersallowed and score
* You will need to get the targets, numbers allowed and score.
* Don’t forget to return them.
* You can complete this by using a data structure such as a list or a stack. You could even use tuples for the target list… This might be easier than a list of objects.
* Think about using a data structure that supports LIFO
* Create a variable showing how many undos are left over.
* Use a list / array to store the score variable in an undo state…

|  |
| --- |
| [Play game method changes]  def PlayGame(Targets, NumbersAllowed, TrainingGame, MaxTarget, MaxNumber):  Score = 0  GameOver = False  #edits start  undoStack = []  maxUndos = 2  while not GameOver:  AddToUndo(undoStack, Targets, NumbersAllowed, Score, maxUndos)  if len(undoStack) > 1:  print(f"{len(undoStack) - 1} undos are available.")  choice = input("Enter 'u' to undo the last move, or press Enter to continue: ").lower()  print()  if choice == 'u':  prevState = UndoLastTurn(undoStack)  prevState = UndoLastTurn(undoStack)  if prevState:  Targets = prevState.GetTargets()  NumbersAllowed = prevState.GetNumbersAllowed()  Score = prevState.GetScore()  continue  #end of edits  DisplayState(Targets, NumbersAllowed, Score)  UserInput = input("Enter an expression: ")  print()  if CheckIfUserInputValid(UserInput):  UserInputInRPN = ConvertToRPN(UserInput)  if CheckNumbersUsedAreAllInNumbersAllowed(NumbersAllowed, UserInputInRPN, MaxNumber):  IsTarget, Score = CheckIfUserInputEvaluationIsATarget(Targets, UserInputInRPN, Score)  if IsTarget:  NumbersAllowed = RemoveNumbersUsed(UserInput, MaxNumber, NumbersAllowed)  NumbersAllowed = FillNumbers(NumbersAllowed, TrainingGame, MaxNumber)  Score -= 1  if Targets[0] != -1:  GameOver = True  else:  Targets = UpdateTargets(Targets, TrainingGame, MaxTarget)  print("Game over!")  DisplayScore(Score) |
| [Creation of method undo last turn]  def UndoLastTurn(undoStack):  if len(undoStack) > 0:  return undoStack.pop()  return None |
| [Creation of method AddtoUndo]  def AddToUndo(undoStack, Targets, NumbersAllowed, Score, maxUndos):  if len(undoStack) >= maxUndos:  undoStack.pop(0)  undoStack.append(UndoState(Targets, NumbersAllowed, Score)) |
| [Creation of new class undo state]  class UndoState:  def \_\_init\_\_(self, targets, numbersAllowed, score):  self.\_targets = list(targets)  self.\_numbersAllowed = list(numbersAllowed)  self.\_score = score  def GetTargets(self):  return list(self.\_targets)  def GetNumbersAllowed(self):  return list(self.\_numbersAllowed)  def GetScore(self):  return self.\_score |
| [print screen of testing] |

10/11 – Undo’s are not incremental in my implementation, though it should have been.