Assignment 9 - Token-Taking Game

Due May 2 by 11:59pm **Points** 20 **Submitting** a file upload **File Types** h and cpp **Available** until May 5 at 12:01am

This assignment was locked May 5 at 12:01am.

ST. CHARLES COMMUNITY COLLEGE

CPT-182 - Programming in C++

Programming Assignment - Token-Taking Game (**20** Points)

(Number in Question Bank: Assignment **9.1**)

Program Overview

In this assignment, you are going to write a C++ program to play a token-taking game. You program should use recursion to find a "path" to reach exactly K tokens within N turns.

Your program **must** be recursive in this assignment; or you will receive **zero** credit.

Rules of the Game

1) At the beginning of the game, you have exactly **27** tokens.
2) In each turn of the game, you may do **1** of the following **2** things:
   - Ask for exactly **39** more tokens.
   - If you have an even number of tokens, you may give back exactly **half** of the tokens you have. However, if you have an **odd** number of tokens, you do **not** have this choice and can only choose to ask for **39** more tokens.
3) The object of the game is to reach exactly K tokens within N turns, where K (positive integer) and N (positive integer) are specified at the beginning of the game.
4) The result of the game could be **1** of the following **2** results:
   - The game reaches a dead end, which means that it is **impossible** to reach exactly K tokens within N turns.
   - A "path" to reach exactly K tokens within N turns is found. Please note that you do **not** need to use exactly N turns. N is the maximum number of turns allowed in this game.
5) For simplicity, you do **not** need to do the following in your games:
   - You do **not** need to find all the paths (sequences) from the starting state to the winning state. Only outputting **1** valid path is sufficient.
   - You do **not** need to count the number of valid sequences.
   - You do **not** need to worry about whether the sequence you found is the shortest path or not.

The `Game` Class

1) The class has the following `private` data fields:
   - **`target`**. This is a `const` unsigned integer representing the number of tokens to reach. You **must** reach exactly this number of tokens to win the game.
   - **`turns`**. This is an unsigned integer representing the number of turns left. This number will decrease by **1** after each move. If the number has gone down to **0** but the current number of tokens is **not** equal to **`target`**, you lose the game.
   - **`path`**. This is a `vector` of `char` that stores the path to reach the winning state. If it is **impossible** to win the game, then the `vector` will be empty.
2) The class has a **constructor** that takes two unsigned integers (representing the target number of tokens and the maximum number of turns allowed) as arguments. The constructor initializes the **`target`** and **`turns`** with the values passed in, and initializes **`path`** with empty `vector`.
3) The class has a function, **`play()`**, that takes an unsigned integer, **`tokens`**, as its only argument. The function returns a `bool`, where a `true` means that a valid path is found and `false` means that a dead end is reached.
   - This function is recursive, which attempts to find a valid sequence to win the game.
   - The first **base case** is that if **`tokens`** is equal to **`target`**, then you already win the game. The function returns `true` immediately.

- The second **base case** is that if variable **turns** has dropped to **0**, which means that you have used up all the allowed turns and still **not** reached the **target**, then it is a dead end. The function returns **false** immediately.
- Then, your function should try to solve the problem. First, you need to try to ask **39** more tokens (denoted as option **A**). Your function should append **'A'** to **path**, decrease **turns** by **1**, and check **play(tokens + 39)**. If it returns **true**, then it means that this choice can lead to winning the game, so your function should return **true**; if it returns **false**, then it means that this choice **cannot** lead to winning the game, so you need to "recover" to the previous state, which means that you need to delete the **'A'** from the rear end of **path** and increase **turns** by **1**.
- Then, if **tokens** is an even number, you need to try to give back half of the tokens (denoted as option **B**). Your function should append **'B'** to **path**, decrease **turns** by **1**, and check **play(tokens / 2)**. If it returns **true**, then it means that this choice can lead to winning the game, so your function should return **true**; if it returns **false**, then it means that this choice **cannot** lead to winning the game, so you need to "recover" to the previous state, which means that you need to delete the **'B'** from the rear end of **path** and increase **turns** by **1**.
- Finally, if both choice did **not** work, then your function should return **false**.

4) The class has a function, **print_result()**, which takes an **ostream** as its only argument and returns **void**. The function first checks whether **path** is empty. If it is empty, then write **"Dead End"** to the **ostream**; it is **not** empty, then write all the characters in the **vector** in order to generate the valid sequence.

Game Examples

target and turns	Expected Output	Explanation
75 6	**AAABBA**	**A**: **27** + **39** = **66** **A**: **66** + **39** = **105** **A**: **105** + **39** = **144** **B**: **144** / **2** = **72** **B**: **72** / **2** = **36** **A**: **36** + **39** = **75** Please note that **ABAAAB** is also a valid sequence, but you only need to find any one valid sequence.
72 2	**Dead End**	It is **impossible** to reach **72** in only **2** turns. The shortest path to reach **72** is **ABA**, which requires **3** turns.

The Input File

- The input file is a **plain text file** (filename: **games.txt**).
- Each row in the input file contains exactly **2** integers, which are the **target** and **turns** of the game (in that order).
- You **cannot** assume (or guess) the number of games stored in the input file. In other words, no matter how many games are stored in the input file, your program should correctly play all of them.
- There may be empty lines at the beginning, in the middle, and/or at the end of the input file. Your program should smartly skip those empty lines.
- Please refer to the **sample input files** to better understand the input file format.

The Output File

- The output file is a **plain text file** (filename: **results.txt**).
- After playing each game in the input file, your program writes the game result to the output file (one result per row).
- Please refer to the **sample output files** to better understand the output file format.

The main() Program

- Parse each game stored in the input file and create a **Game** object.
- Play the game by calling the **.play()** function.
- Write the game result to the output file by calling the **.print_result()** function.

Sample Input and Output File (Click to Download)

Sample Input File 1 (https://drive.google.com/uc?export=download&id=1sFLUvAiL5DSveUfYlKNPjkVZOp-5w_jF) **Sample Input File 2** (
Sample Output File 1 (https://drive.google.com/uc?export=download&id=1s4pP4Oss3nElQF_KaScvhq6lGPB72gYD) **Sample Output File 2**

Assignment Submission and Grading (Please Read)

- Please upload all your **.h** (if any) and **.cpp** files (**not** the ~~entire Microsoft Visual Studio project folder~~) on Canvas.
- Before the assignment deadline, you can submit your work unlimited times. However, only your latest submission will be graded.
- At least **20**% of your code should be **comments**. All variable, function (if any), and class (if any) names should "make good sense". You should let the grader put **least effort** to understand your code. Grader will **take off points**, even if your program passed all test cases, if he/she has to put extra **unnecessary** effort to understand your code.
- Please **save a backup copy** of all your work in your computer hard drive.
- Your program will be graded (tested) using another valid input file (still named **games.txt**) to check whether it can generate the expected (correct) output file (with correct format and correct output values in it). As long as the input file is valid, your program should generate a correct output file. In other words, your program should work for **any** valid input file, **not** just the sample input files provided in the assignment instructions.
- In this class, you can assume that the input file (input data) is always **valid** and **has correct format**. You do **not** need to deal with ~~invalid input~~ or ~~error handling~~.
- Your work will be graded after the assignment deadline. All students will receive their assignment grades at (almost) the same time.