Programming Project 12

Due: Wednesday, 4/17/13 at 11:59 pm

AI Guessing Game using Hash Tables

For this program, the computer will try to predict whether the user will guess "heads" or "tails". The AI algorithm relies on the fact that while a person thinks they are coming up with random guesses, they are actually following a pattern even if they don't realize it. The AI algorithm will remember user's patterns and use this to predict the user's next guess.

The program/user interaction is as follows:

1. The user needs to pick either heads or tails.
2. Before the user makes his/her pick the program will make a prediction as to whether the user is going to pick head or tail.
3. The programs prediction is hidden from the user until he makes his actual pick.
4. Once the user makes his pick the programs prediction is revealed.
5. If the program correctly predicted the users pick it gets a point otherwise if the program misses the user gets a point.
6. Whoever (program or user) gets 25 points first is deemed the winner.

Below is a possible sample of the input/output of the program. The users guess (either h or t) is shown in bold type.

```
Welcome to MindReader

Guess head or tails and I'll predict your guess.
What is your guess [h/t] ? t
Oh no. I predicted h
Score = 0 | 1

Guess head or tails and I'll predict your guess.
What is your guess [h/t] ? h
Yes! I too predicted h
Score = 1 | 1

Guess head or tails and I'll predict your guess.
What is your guess [h/t] ? h
Oh no. I predicted t
Score = 1 | 2

Guess head or tails and I'll predict your guess.
What is your guess [h/t] ?
...
```

The hash table will use need to store 3 values at each item in the table:

- a 4 character string specifying the last 4 guesses made by the user
- the number of times the user guessed heads after this specific pattern of 4 guesses
- the number of times the user guessed tails after this specific pattern of 4 guesses

The key of the hash table will the 4 character string. The key is the item on which hash function is applied.

The idea behind the AI algorithm is that a person is likely repeat a 5th guess after a specific pattern of 4 guesses. The computer will keep track of the user's last four guesses, look up this pattern in the hash table and use the information stored to predict whether the user's next guess will be heads or tails. If the data stored in the hash table regarding the last four guesses indicates more heads than tails have been guessed by the user, the computer will predict heads. If the data stored indicates more tails than heads have been guess, the computer will predict tails. If the user has made the same number of heads and tails or if this pattern is not contained in the hash table, the computer will make a random prediction.

For the AI algorithm to work, there must be enough guesses made to establish some pattern. This is why the game is being played to 25 points. If played to a smaller number of points, the computer will pretty much be making random predictions.

The high level structure of this program is:

1. The computer makes a prediction based on the pattern of the last 4 user guesses from the information stored in the hash table and conceals it from the user.
2. The user makes a choice/guess and enters it via standard input.
3. The computer updates the score appropriately as to whether it made a correct prediction or not.
4. The computer stores the user's choice/guess in the hash table entry for the pattern of the previous 4 user guesses by incrementing the number of times heads or tails was guessed for that pattern.
5. If 25 points has not been reached by the computer or user, go to step 1 updating the pattern with the most recent guess. If 25 points has been reached, print a message declaring who won.

Let us assume the user made the following guesses:

1. heads
2. heads
3. tails
4. heads

The four character string created by this pattern would be: hhth

Now let us assume the user's next guess is tails. In this case, the item in the hash table with key of hhth will increment by 1 the number of times tails was guessed for this pattern.

Now the four character string based on the last four guesses would be: htht

Note that the first time a pattern is encountered, the hash table will not contain an entry for that pattern. Your program will then need to add this pattern into the hash table. Also note that for the first four guesses, there have not been enough guesses made to build a four character string of previous guesses. In this case, either

Your program is to display the entire hash table at the end of the game when 25 points has been reached and whenever the user enters a value of "d" when being asked for a guess of heads or tails. Thus at the prompt of:

```
What is your guess [h/t] ?
```

your program has four options to check for:

1. a value of "h" for heads,
2. a value of "t" for tails,
3. a value of "d" to display the entire hash table,
4. or some other value that should result in an error message.

When displaying the hash table, you must list the position in the array where the item is stored along with the pattern and the number of heads and tails guessed with that pattern. The size of the hash table is left up to you but is should be some value of 10 or greater. The hash function is also left up to you but should result is some nearly uniform distribution of the patterns.

MULTIPLE SOURCE CODE FILES

Your program is to be written using at least two source code files. It must also have a makefile to help with the compilation of the program. All of the hash table code is to be in one source code file. The other code is to be in a different source file (or in multiple different source files if you want to have more than two source code files for the project). This information was first discussed in the lab exercise 9. Information from lab exercise in is listed below.

You must also create a header file. The job of the header file is to contain the information so the source code files can talk to each other. The header file (.h file) should contain the function prototypes and any struct and/or typedef statements. Please review the .h file in the example below.

The makefile MUST seperately compile each source code file into a ".o" file and separately link the ".o" files together into an executable file. Review the makefile in the example below to see how this is done. The command to create the .o file is:

```
gcc –c program1.c
```

The command to link the files program1.o, program2.o and program3.o into an executable file is:

```
gcc program1.o program2.o program3.o
```

The above command will just name the executable file using the default name, most often the –o option is given to provide a specific name for the executable file.

```
gcc program1.o program2.o program3.o –o program.exe
```

Example of Multiple Source Code Files

Consider the program contained in the following files:

- max3a.c
- max3b.c
- max3.h
- makefile

This example shows how to set up this simplest of multiple source code file program. Note that max3a.c and max3b.c just contain functions and a #include of max3.h. The file max3.h contains the prototypes (or forward declarations) for all of the functions that are called from outside its source code file and any "globally" needed information.

The makefile is a special file that helps in the compilation of the source code into the object files into the executable file. A makefile is executed by the use of the **make** command. The syntax of a makefile can be strange. I have always found that it is easiest to modify an existing makefile rather than trying to create one from scratch. The makefile will contain multiple rules. Each rule has the following syntax:

```
target:  dependencyList
         commandLine
```

The multiple rules in the make file are separated by a blank line. Also note (this is VERY IMPORTANT) the commandLine must use a TAB for its indentation! An example of a rule is:

```
max3a.o: max3a.c max3.h
      gcc -c max3a.c
```

The **commandLine** is **gcc -c max3a.c**, which will compile the source code file of max3a.c into the object code file of max3a.o.

The **target** in the above example is the file **max3a.o**, which is also the name of the file created when the commandLine is executed. This relationship is what causes makefiles to work.

The **dependencyList** in the above example is the two files: **max3a.c** and **max3.h**, which are the files needed for the commandLine to properly run. Again, this relationship is what causes makefiles to work.

The make command uses the timestamps of the files in the target and the dependencyList. If any file in the dependencyList has a more recent timestamp than the target file, the commandLine is executed. The idea is that if the user has recently changed either **max3a.c** or **max3.h**, then the object file **max3a.o** needs to be re-compiled. Make is designed to help the programmer keep track of what needs to be compiled next.

Make and makefile tutorials can be found at:

- http://mrbook.org/tutorials/make/
- http://www.gnu.org/software/make/manual/make.html
- http://www.opussoftware.com/tutorial/TutMakefile.htm

Program Submission

You are to submit the programs for this lab via the Assignments Page in Blackboard.

To help the TA, name your file with your net-id and the assignment name, like:

- ptroy1LabX.c