Final Project List

CSI 31. Luis Fernandez

Deadlines

Specification and Program Outline: Thursday April 14 at 12:30pm (to be sent by email to luis.fernandez01@bcc.cuny.edu)
Progress report and initial working program: Friday May 6 at 11pm. (to be sent by email to luis.fernandez01@bcc.cuny.edu)
Project Due Date: Monday May 23 at 11pm

- Do ONE of the projects listed below (if you want to do something else, you must check with me now!). Work on your own, without help from others, whether a person, someone online, or by copying code from some source.
- For all projects, the first part consists of typing up a detailed and clear specification, as well as an outline of the program. The projects are somewhat open-ended and not fully specified; in your specification, you should explain exactly what you want to do.
- The progress report should consist of a working program where the core of the project is already working, and only additions and modifications are needed to complete the final project.

Grading rubric

- 45 to 75 points (depending on the difficulty of the chosen project): the program works well and does what it is supposed to do
- 10 points: deadlines are met.
- 10 points: well-commented (inner comments are present, correct and meaningful).
- 10 points: clear input/output and intuitive interface
- 10 points: reasonably fool proof.

Project Descriptions

Easy:

1. Write a program that receives a list of integers separated by spaces from a file (or from the keyboard) and outputs a file (or prints out in the terminal) with the integers sorted in increasing order. You will have to first read the input and store it in a list. Then define a function `simpleSort` that takes a list as input and sorts it.

   The algorithm works as follows: if the list has size n, the algorithm does $n-1$ passes over the list. In each pass, it starts from the first element and checks if it is greater than the second. If it is, it swaps the values, otherwise it does not change them. Then checks if the second element is greater than the third and swaps them if it is. Then checks the third and the fourth, etc., until it reaches element $n-1$ and compares it with element n.

   The pseudocode for the algorithm in the function `simpleSort` is as follows:

```
def simpleSort(list):
  for i from 1 to n-1
    for j from 0 to n-i
      if list[j] < list[j+1]), swap  list[j] and list[j+1]
}
```

In this program, the user should be first asked whether the numbers should be read from a file or from the keyboard, and whether the output should be to the screen or to a file. Then the user must enter either the name of a file for inputs or the list of numbers, and the name of a file for output, if it applies.

2. Counting letters program

   Write a program that finds the number of occurrences of letters from the alphabet (ignoring lower case-upper case) in a file (the file name is given by the user), then outputs the two-column list with result. The first column is a list of letters of the alphabet; the second column is the count of the number of occurrences of each letter from the alphabet. The two-column list should be sorted on entries in the first column, i.e. it starts from letter "a" and ends with the letter "z".

   Example:

   Input data (taken from a file): Hello, my name is Luis Fernandez.

   Output: letters of the alphabet occurrence

Integer values	Occurrences
a	2
b	0
...	
d	1
e	4
...	

Medium:

3. Shooting range

   The goal of this project is to do a shooting range game. The skeleton of the program is drawing a "gun" (could be just two rectangles together, you can design what you want) that can be rotated (using the arrows in the keyboard for example) to aim to a target at the other end of the window. When a key is pressed, the gun shoots a bullet (which can be a small rectangle, or a short line, or an oval...).

   The position of the target should be random at the beginning of each game (otherwise it is too easy!). You can have several targets that disappear when they are hit, or one target with different scores (small part in the center with highest score, and lower scores as you go out of the center of the target).

   Every time the target is hit, the program should give a score (either how many targets hit so far, or points if the target has different scores, etc).

   From here you have to be creative. Here are some ideas.

- To make the game more challenging, you can add an option "wind" so that the bullet drifts to one side slightly. The drift should be fixed at the beginning of each game and should be random.
- The position of the gun can also made random at the beginning of each game.
- You can try to implement moving targets.
- Be creative!

4. Paddle racketball game

   The goal is to make a racketball game where you have a "paddle" (a rectangle will do) that bounces a ball against the walls on the top and sides of the window. There should be a counter with the number of times that the ball was hit. The initial angle of the moving ball should be random, and shooting up. Some ideas:

   - Make the direction of the ball change when the paddle is moved right when it hits the ball.
   - Make the direction of the ball change when it hits the corners of the paddle.
   - To make it more challenging, reduce the size of the paddle when a number of hits is reached.
   - Be creative!

Advanced:

5. Multiple-Pile Nim

   Nim is a 2-player game. There are some number of piles (say p piles) each with some number of objects (say each pile has n objects). Play alternates between the players. On a turn, a player may remove **any number** of objects (at least one) from any **one** of the piles. The player to remove the last objects wins.

   Design Nim for the computer (*if you want to do this using graphics, you must first do a text interface version and submit it to me, receive the okay from me, and then work on one that uses graphics*). First the program should prompt the user for the "p" and the "n" (you can require that p is at most 5 and n is at most 20, especially for a graphical version). Then it should prompt the user for the option of a two player game or a one player game (in the one player case, the other player is the computer). Then play commences. The program should be fool-proof and easy to use, for example:

   - If a player wants to remove no objects, the program should object (i.e. it should be fool-proof!).
   - After each move inform the user whose turn it is and what the state of the piles are (i.e. it should be easy-to-use!).
   - Etc.

   Final Feature: *Once the above is working, add ONE of the following features*:

(a) When played against the computer design it so that the user is prompted for difficulty levels (the hardest should be one which always wins when it can; there is a known best strategy in Nim and this will require understanding that strategy).

(b) Add a feature that allows the computer to play itself. Then design various strategies (at least 3). Then pit the various strategies against one another in hundreds to thousands of games to see how the strategies compare. You must describe the different strategies and describe the outcome of your experiments.

6. Crytopgraphy

Write an *encoder* and *decoder* (as we did in class) but instead of the key being a shift, it should be a rearrangement of the 26 letters of the alphabet indicating how to do the substitution in the encoding. The encoder and decoder should take a file as input and create a file as output. For example if the key starts out:

$$BSYJA...$$

then "ACE" would be encoded as "BYA".

Also write a *code cracker*: It takes a coded file (and **no key**) and tries to decode the file using frequency analysis. To decode, it should take the file and an ordered list of the frequency of letters in English (call this the *frequency list*), starting with most frequent letter. The program should use the frequency list to try to crack the code by matching the frequency of letters in the coded file with the frequency list. For example, suppose the frequency list is as follows (middle missing):

$$EIS \dots XZ$$

Then the most frequent letter in the coded file would be replaced by E, the second most frequent by I, and so on. To emphasize: The frequency list is one of the inputs to the function; you do not need to find it.

Once you have written the encoder, decoder, and code cracker, run experiments to see how good your code cracker is. Do this by: Taking longer texts, encoding them, and then running the code cracker on them; try different frequency lists and look up established frequency lists.

The program should be fool-proof and easy to use, for example:

- If the key is too short or repeats letters, the program should ask for a new key (i.e. it should be fool-proof!).
- In using the program, it should clearly direct the user on what to input at each step (i.e. it should be easy-to-use!).
- Etc.

Final Feature: Write another piece of code to determine a frequency list experimentally; call it the *frequency finder*. The frequency finder should read in a file and then output a frequency list of the 26 letters of the alphabet. Run the frequency finder on many files to come up with a frequency list, then use this frequency list on coded files to see how it works.