Name: Sylvia Le

Course: COM110

**WordCloud Design**

1, Top-Level Design

* Identify 3 main jobs: input, process and output.

+ Input: get the file name the user type in

+ Process: start counting for frequency and sorting based on frequency

+ Output: display the results on the screen

* Given those three jobs, the main() can just have: create a graphic window, getInput(), process() and displayOutput()

2, Second-Level Design

* The getInput() function can be tackled first: draw the name of the program, entry box and a button on the graphic window. Have the function returns the file name that the user type in upon the button click
* Upon creation of the function, two other helper functions must be added: drawButton() and isClick(), in order to make the code more compact and to check if the user click the button
* The reading (by professor Zelle) has already ‘sort of’ provide the processing function (named analyze()). The creating of this function need help of another function called byFreq(), to sort the result by frequency
* The displayOutput() can now just be printing the list of words on the shell, to make sure the two other functions work OK
* After this step, I notice: the analyze() call the getInput() function to get the file name, then the displayOutput() (later named WCGraphics), call the analyze() function to get the list of the words to display

→ just need to call WCGraphics() from main()

3, Third-Level Design

* Remove punctuations and stop words from the text. Since there is (quite) a lot of stop words, I got a txt file from GitHub contains a list of stop words, read the file and split it into a Python list of stop words, the do the checking.
* Start displaying the words, with random color and random position on the graphic window. Ignore the collision for now

4, Fourth-Level Design

* Start thinking about the collision avoidance problem. I came up with 3 ideas (the last one was actually suggested by Professor Tarimo). Below is the brief explanation of the ideas

1. Idea 1:

* Imagine drawing a grid of rectangles on the graphic window, and keep a list of the coordination of the two corners of those rectangles
* Pick random points, base on the list of coords above, check which rectangle the current point is in, then append that rectangle to an ‘omit’ list
* Continue choosing the points, making sure that each rectangle only contain one points, by checking with the omit list
* Problem and reason for not choosing this approach:

+ Got a lot of errors on the use of lists

+ Cannot effectively minimize overlap (imagine if the random point chosen is very close to the margin of the rectangle, then it will still overlap with the word in the neighbor rectangles)

1. Idea 2:

* For each word that need to display, choose a random point. Using some linear functions (will explain below), calculate the coords of the two corners of the rectangle that entrap the word inside. Check if the rectangle overlap other rectangle that entrap the words inside; if yes, start finding again; if no, draw the words
* The math of creating rectangles:  
  + Suppose each letter fit in side a square (instead of rectangle, in reality). (1)

+ From the web (<https://websemantics.uk/articles/font-size-conversion/>), with the provided number, I can create a system of equation to calculate the slope and the y-intercept of the function y = ax +b, in which x is the font size of the word (this is determined base on the the word’s frequency), and y is the height of the word in pixel

The final function (after rounding) is y = 1.5x – 1

Then I half the result to get the number that I have to add/subtract to the coords of the random point chosen

+ As suppose (1), the length of each word in pixel will be the number of letters in that word multiply by y (in the function above).

Then I half the result, and then divide by 1.2 (because the size of the graphic window is 800-600 pixel, but I setCoords (0, 0, 600, 600))

* Checking rectangles overlap: testing multiple suggestions on Stack Overflow, and choose the ost compact
* Evaluation:

+ Pros: maximize the randomization; and if works then will be a very effective way to avoid collision

+ Cons: Depends (heavily, I guess), on the screen size and the resolution of the computer (I tried on Professor Tarimo’s Mac and the rectangle seems a little to big for the words

* Reason for not choosing:

+ For reason that still remains unknown, words still collapse, like, a lot.

+ However, I still attach the file that follow this idea in the package (wordcloud (2).py). And if I will really really appreciate it if you can tell me what went wrong T.T

1. Idea 3:

* Create a grid of points (can be considered slots) on the graphic window, keep track of it in a list
* For each word that needed to display, choose a random point from the above list. The chosen points will be appended to another list (omit list).
* If the point chosen has already been chosen before (i.e. that point is in the omit list), choose again.
* After choosing, begin setting size and random color for the word, then draw it on the graphic window
* Quick evaluation

+ Does quite a good job on collision avoidance (if the number of display words increase and the size increase then if the big words is close to one another, there will still be overlap)

+ Display is… not so random. Words are still quite aligned

5, Fifth-Level Design

* Add some features

+ A message box that tell users what the program does before showing the entry box, button and stuff

+ Set the length (in words) of the text file as the number of word display, if that length is smaller than 30

+ A message box to tell the user how to quit (as the design is quite, well, chain-ish, so I really don’t know where to insert the code for the quit button)