David Barnett

Alberto Salmeron

Lab 9

- We would use a Binary Search Tree by inserting the line number of all lines of code being tested. As each line appeared in the tests, we would check if the line number remained in the tree. If so, then we would remove that line number. Once the tests run, we would report an infix list of the line numbers still in the tree.
- We would use a Hash Table by adding all line numbers in the tested file as nodes. As each line appeared in a test, we would mark the node as visited. Once the tests finished, the line numbers would be revisited through their keys to check for unvisited lines, and output these lines as a list.
- We can use a Stack to keep track of which line is being checked. All lines get pushed into the stack. A line is popped and tested to see if it's used. If used, the next line gets popped, else that line is pushed onto a different stack that contains the lines that weren't used.

The README for the code has some information on the code's purpose, as well as versions of python it runs on, and information for download and support. The majority of the files are source code, part of which is a test folder that contains plenty of tests. Generally, the code is heavily commented, making it easy to tell each functions purpose. The comments are often too in-depth, however, so some comments are oversaturated with unnecessary information, masking the helpful information. Many functions in the code are only several lines that often call other functions.

data.py- The data file collects executed lines, source arcs, file tracer names, and other information about the execution of the program. This file uses hash tables to store the information for the aforementioned elements. The line table takes the data from the file, and maps the file names to a dictionary. The arc table does a similar operation, but instead of using the line data, it uses the arc data. The file tracer table stores the file name with the plugin name. Each step and function is commented well enough to understand what is happening, therefore maintaining and updating the code seems like it would be easy enough. This code contains more comments than our code would. It also uses more advanced methods of doing certain things.

collector.py- This file creates a collector for all the tracers using the data. It uses a stack to determine which collector is presently active. All collectors are pushed, and any collector that's not popped is paused. Hash tables are used to map the file names to the line number keys, and the file names to tracer plugins. The comments and the code make it clear what's going on. Maintaining and updating this code would be easy. Besides the surplus of comments, the code is similar to our style of code.

results.py- The results file finds the coverage percentage and missed lines. This file uses a hash table once, but there are no other data structures present, with the authors relying heavily on sorted lists. The function with the hash table uses it to output the lines of code that were not covered in its test file. With the exception of the __init__ function, each function is commented and formatted well, making it clear and readable. The __init__ function, however, is cluttered, making it difficult to see the purpose of each line. Besides that function, however, we would be willing to update and maintain the code, and the code looks similar to how we would write it.

summary.py- The summary file contains the code for the output the user sees once the code has run. Additionally, if there are errors in the file, it will show error messages rather than the anticipated output. This file does not use any data structures we have learned, but rather writes and formats output from lists from the report file. The code produces a well-created output and the comments are helpful, but the majority of the code is in one function and it has relatively few comments compared to the other files, which makes it difficult to see the purpose of each individual line. Therefore we would have difficulty updating or maintaining the file. This code differs from how we would make it mainly in that we would divide the large function into many smaller functions.