The University of Texas
Computer Science

313E Programming Assignment 05
Working Hard

Setup

This assignment may be completed individually or with a pair programming partner. Be sure to include your and your partner's name and EID in the Python file header. If you are working alone, you may delete one of them.

Complete these steps to prepare for the Assignment.

Check	Description
▢	Download work.py, work.in, and work.out. You will be working on the work.py file.
▢	Place all three files in the same folder/directory.
▢	You may not change the file names. Otherwise, the grading script will not work.

work.out is a sample of what your output should look like when your program is complete. You do not have to edit this file. **work.in** is the file that your program will be reading from.

If you would like to turn on autosave in VSCode, click on **File -> Autosave.**

Problem Description

Chris has to complete a programming assignment overnight. She has to write many lines of code before morning. She is dead tired and she wants to drink coffee to stay awake. Each cup of coffee helps her stay awake, but it also makes her jittery and she writes code slower. If she drinks too much coffee, she can't write any code at all. But, she wants to drink coffee as soon as possible, because feeling that tired is a horrible feeling. She wants to know - what is the minimum number of lines of code she needs to write before the first cup of coffee, so she can finish the code before her productivity reaches 0 and she falls asleep.

Requirements

Write a program that will help Chris calculate how many lines of code she needs to write before drinking coffee so that she can finish the code before her productivity reaches 0 and she falls asleep. You will modify each function according to each function's purpose.

1. Your program uses the text file **work.in** provided to create your final output. You will be using input redirection to read from this file.
2. **You may not change the names or parameters of the functions listed. They must have the functionality as given in the specifications. You can always add more functions than those listed.**
3. **You must implement linear_search() using linear search and binary_search() using binary search.**
4. **You may not import any additional external libraries in your solution besides time.**

You will modify the following functions to implement the functionality below.

sum_series(lines_before_coffee, prod_loss)
This function will determine how many lines of code will be written before the coder crashes to sleep. It will take in two arguments, lines_before_coffee and prod_loss. **lines_before_coffee** will be the number of lines that the coder has written before drinking coffee, and **prod_loss** is the factor for loss of productivity after drinking coffee. This function will return the number of lines of code that will be written before the coder falls asleep.

linear_search(total_lines, prod_loss)
This function will use a linear searching algorithm to find the initial lines of code that must be written to ensure that the coder completes the total lines of code before the first cup of coffee and falling asleep. **total_lines** will be the total number of lines of code that have to be written, and **prod_loss** is the factor for loss of productivity after each cup of coffee. This function will return the initial lines of code that need to be written before the first cup of coffee and the number of calls to sum_series as a tuple.

binary_search(total_lines, prod_loss)
This function will use a binary searching algorithm to find the initial lines of code that must be written to ensure that the coder completes the total lines of code before the first cup of coffee and falling asleep. **total_lines** will be the total number of lines of code that have to be written, and **prod_loss** is the factor for loss of productivity after each cup of coffee. This function will return the initial lines of code that need to be written before the first cup of coffee and the number of calls to sum_series as a tuple.

main()
This function will handle the input files using input redirection, gather efficiency data, and print the output for you. You will need to update this function to read the correct number of cases, as well as reading each subsequent line to read information for the total_lines and prod_loss. **You will not need to modify anything else**.

Example 1

Total lines of code to write: 199
After coffee, productivity drops by factor of: 2
Initial lines of code (how many lines of code before first cup of coffee): 20

Round	New Lines of Code		Lines of Code Written
1	20	Drink Coffee	20
2	20 // 2 = 10		30
3	20 // 4 = 5		35

4	20 // 8 = 2		37
5	20 // 16 = 1		38
	20 // 32 = 0	STOP	

In this example, Chris reaches 0 productivity before she finishes the total lines of code. She will need to write more lines of code before having her first coffee.

Example 2

Lines of code to write: 199 (same as example 1)
After coffee, productivity drops by factor of: 2 (same as example 1)
Initial lines of code (how many lines of code before first cup of coffee): 120

Round	New Lines of Code		Lines of Code Written
1	120	Drink Coffee	120
2	120 // 2 = 60		180
3	120 // 4 = 30		210
4	120 // 8 = 15		225
5	120 // 16 = 7		232
6	120 // 32 = 3		235
7	120 // 64 = 1		236
	120 // 128 = 0	STOP	

In this example, Chris completes all 199 lines of code before reaching 0 productivity. But, it is possible that she could have had her first cup of coffee sooner, maybe after writing 100 or 115 lines of code. She wants to know what is the least number of lines of code she can write initially so she can drink coffee as soon as possible, and still complete all the code before reaching 0 productivity.

Input

Your input will be from a file called work.in. Do not hard code the name of the file in your program. You will need to use input redirection to run this program using **work.in**. The input redirection will be handled by the shell (terminal); you will not have to implement this in your code.

You can run the program by typing the following command into your terminal:

```
python3 work.py < work.in
```

Alternatively, if you are using a Windows computer and are having trouble with input redirection <, you can run:

```
cat work.in | python3 work.py
```

Here is an example of an input file:

```
3
300 2
59 9
8000 5
```

The first line of input files is the number of scenarios to analyze. This will be followed by one line for each scenario. Each line will have two numbers. The first number will be the total number of lines of code to be written (between 1 and 1000000) and the second number is the productivity loss factor (between 2 and 10).

Input validation:

- The code can assume that the file is structured properly, that all tokens are integers and that all integers are within the specified range.

Output

In your output, each scenario will be separated by a header. Each scenario includes the following, with appropriate labels:

For binary search and then linear search:

- The ideal number of lines of code to write before the first cup of coffee
- How many times the sum_series method was called
- How many seconds it took to find the solution

The last line of output states how much faster the binary search solution was. The exact elapsed time will vary by code, machine and other factors. Your code will **only** be graded by matching format and the ideal number of lines of code for each case. You do **not** have to match the number of calls to sum_series(), the elapsed time, or how much faster binary search is. Your number of calls to sum_series() for binary search **must** be O(logN).

The output for the provided input file is below.

```
=====> Case # 1
Binary Search:
Ideal lines of code before coffee:  152
sum_series called 13 times
Elapsed Time:  0.00005198 seconds

Linear Search:
Ideal lines of code before coffee:  152
sum_series called 152 times
Elapsed Time:  0.00038600 seconds

Binary Search was 7.4 times faster.
```

```
=====> Case # 2
Binary Search:
Ideal lines of code before coffee:  54
sum_series called 7 times
Elapsed Time:  0.00001192 seconds

Linear Search:
Ideal lines of code before coffee:  54
sum_series called 54 times
Elapsed Time:  0.00005412 seconds

Binary Search was 4.5 times faster.


=====> Case # 3
Binary Search:
Ideal lines of code before coffee:  6401
sum_series called 18 times
Elapsed Time:  0.00005198 seconds

Linear Search:
Ideal lines of code before coffee:  6401
sum_series called 6401 times
Elapsed Time:  0.01395011 seconds

Binary Search was 268.4 times faster.
```

Approach

It might not seem obvious at first, but this problem can be seen as a search problem. Given the following scenario from the input file, the code needs to determine the ideal number of lines of code to write before drinking the first cup of coffee in this case:

- Total lines of code to be written: 300 (total_lines)
- Productivity loss factor: 2 (prod_loss)

Related variables are:

- Initial lines of code to be written before first cup of coffee (initial_lines)
- Total lines that would be written before sleep, based on intial_lines (total_for_initial_lines)

Chris would like to have coffee as soon as possible, because she is tired. If she drinks coffee after writing 1 line of code, then she will write 1 // 2 lines of code after that, which is 0, and she will fall asleep before completing the work.

If Chris writes all 300 lines of code before the first cup of coffee, she will certainly complete all of the work. But, this is not the ideal solution, since she wants to drink coffee as soon as possible, and still complete the work.

The ideal initial number of lines of code to write before the first cup of coffee is somewhere between 1 and 300, inclusive. If you know how many lines of code can be finished for each number 1 - 300, then you could do a search of these results to find the ideal initial lines of code to write.

What data structure would you use to store the possible initial lines of code (1-300) and the resulting lines of code?

How would the index work for this data structure?

Draw the data structure with sample data.

If the data structure was filled in with all data, how would you find the ideal initial lines of code manually?

How could you find the same thing with a linear search?

How could you find the same thing with a binary search?

How could you end each of the above searches early, as soon as the ideal initial lines value is found?

Note: On fast computers, the linear search and binary search methods may execute so quickly, that no measurable time has elapsed.

Grading

Visible Test Cases - 75/100 points

The program output exactly matches the sample solution's output. To receive full credit, the program must produce the correct output based on all requirements in this document and pass all the test cases.

Hidden Test Cases - 15/100 points

The program output exactly matches the sample solution's output. To receive full credit, the program must produce the correct output based on all requirements in this document and pass all the test cases.

Style Grading - 10/100 points

The wordle.py file must comply with the PEP 8 Style Guide.

If you are confused about how to comply with the style guide, paste the error or the error ID (e.g. C0116 for missing function or method docstring) in the search bar here in the Pylint Documentation, and you should find

The TAs will complete a manual code review for each assignment to confirm that you have followed the requirements and directions on this document. Deductions will occur on each test case that fails to follow requirements.

Submission

Follow these steps for submission.

Check	Description
▢	Verify that you have no debugging statements left in your code.
▢	Submit **only** work.py (the starter code file with your updates) to the given assignment in Gradescope. This will cause the grading scripts to run.
▢	If you have a partner, make sure you are submitting it for you **and** your partner on Gradescope. See this Gradescope documentation for more details.
▢	When the grading scripts are complete, check the results. If there are errors, evaluate the script feedback, work on the fix, test the fix, and then re-submit the file to Gradescope. You can submit as many times as necessary before the due date. NOTE: By default, only the most recent submission will be considered for grading. If you want to use a previous submission for your final grade, you must activate it from your submission history before the due date.

Academic Integrity

Please review the Academic Integrity section of the syllabus. We will be using plagiarism checkers, and we will cross-reference your solution with AI-generated solutions to check for similarities. Remember the goal in this class is not to write the perfect solution; we already have many of those! The goal is to learn how to problem solve, so:

- don't hesitate to ask for guidance from the instructional staff.
- discuss with peers about **only** bugs, their potential fixes, **high-level** design decisions, and project clarifying questions. Do not look at, verbalize, or copy another student's code when doing so.

Attribution

Thanks to Dr. Carol Ramsey and Dr. Kia Teymourian for the instructions template and this assignment.