

This project is worth zero points, but is **required** in order to receive grades on the rest of the projects. Its purpose is to ensure that you are prepared to use Gradescope to submit projects for this class. It exercises folder structure, command line access, opening and reading from/writing to files, and so on, aspects of all of the projects in this class. (Even though we say it is worth zero points, it will show as being worth 1 point in Gradescope, but that's just so we know who has completed it to the point that it works.)

The program will be submitted to and graded by the autograder within Gradescope. You will write a program called **setup** that will accept command line arguments as follows:

```
setup inputFile.gz outputFile k
```

Your *setup* program will open up *inputFile.gz* (a gzip-compressed file) for reading. It will also open up *outputFile* for writing; if *outputFile* already exists, it should be **silently overwritten**.

Your *setup* program will then read through every line in the file, break the line into tokens by whitespace, and print the k^{th} token of that line to *outputFile*. If there are fewer than k tokens on a particular line, print "*Too Short*" for that line.

When all of the input lines have been processed, print one final line to *outputFile* that contains three things:

1. The total number of lines in *inputFile.gz* (where the count includes lines that are too short)
2. The minimum and maximum tokens by alphabetical order (normal comparison, so case sensitive and punctuation sorted typically) that were printed out, except *Too Short* is not included in that list being compared. If there are no tokens found, print *null* for *both* the minimum and maximum.

As an example, if the input text (after uncompressing) is in a file called *p0-sample.txt.gz*:

```
The west wind whispered, And touched the eyelids of spring: Her eyes, Primroses.
Whitecaps on the bay: A broken signboard banging In the April wind.
beneath leaf mold stone cool stone
an icicle the moon drifting through it
```

Then the command line "*setup p0-sample.txt.gz p0-output.txt 7*" would produce the following output in the file *p0-output.txt*:

```
the
signboard
Too Short
it
4 it the
```

(The haikus above are taken from the Wikipedia article called Haiku in English. Attributions of the haikus are given there.)

To help you debug your code, we are providing the following files (see the bottom of this page):

- P0-sample.txt.gz is an example input file with several lines that is compressed
- P0-sampleout.txt is the expected output for that input
- P0-test.txt.gz is an input file that you will use to test your program and that the autograder will also use to test your program
- P0java.zip is a skeleton of the zip file you will be uploading, assuming you are programming in Java
- P0python.zip is a skeleton of the zip file you will be uploaded, assuming you are programming in Python

You will submit a single zip file that includes all of your files in the following structure:

- README.txt is a text file that includes the amount of time you spent on this project and any thoughts about this startup project that you wish to share. See the sample zip file for the format of this file.
- P0-testout.txt is the output of your program on P0-test.txt with $k=2$
- src/ is a folder that contains your source code. Do not include data files, class files, or anything that is not part of your program. For P0 it includes:

◦ `setup.java` or `setup.py` is your program. Note that it must be within the `src/` folder, not at the same level as the other files. The structure is important and you cannot pass the autograder test without getting it right. You may prefer to start with `P0java.zip` or `P0python.zip` to ensure your submission is in the correct format. You then just upload the zip file to Gradescope (it seems to be nearly impossible to get a folder structure otherwise).

At submission time, we will run `P0-sample.txt.gz` file through your code with `k=7` and give you immediate feedback so you know that your code works in the autograder (you presumably will be confident that it works because you have a solution!). We will also run `P0-test.txt.gz` through your code with some value of `k` and give you immediate feedback on whether that works. We may run other tests to exercise your code.

To help you get started, we are providing some sample code (in the zip files) described below. The sample code fragment we provide and discuss below will crash with bad input values. You may prefer to modify your code to more gracefully handle errors in the commands, the input file, or anything else. Doing so will help you debug your code and may make some autograder output easier to understand.

Coding for the autograder in Java

Your code will be tested in a JDK 17 environment. Put all of your code (which for P0 is probably one file) in the `src` folder. You should have, at a minimum, `setup.java` which includes a `main()` method that can run the word counting on an input file.

We strongly recommend that you make your life easier by making the command line arguments optional and providing sane defaults. Here is some code that shows one way to accomplish that (this code is also provided in `setup.java` provided in the sample zip file). It lets you use default arguments for all arguments left off, including – if there are no arguments at all – running the program as if the command line above had been typed.

```
public static void main(String[] args) {
    // Read arguments from command line; or use sane defaults for IDE.
    String inputFile = args.length >= 1 ? args[0] : "P0-sample.txt.gz";
    String outputFile = args.length >= 2 ? args[1] : "P0-sampleout.txt";
    int k = args.length >= 3 ? Integer.parseInt(args[2]) : 7;
```

Coding for the autograder in Python

Your code will be tested in a Python 3.10 environment. Put all of your code (which for P0 is probably one file) in the `src` folder. You should have, at a minimum, `setup.py` which includes a `main()` method that can run the word counting on an input file.

We strongly recommend that you make your life easier by making the command line arguments optional and providing sane defaults. Here is some code that shows one way to accomplish that (this code is also provided in `setup.py` provided in the sample zip file). It lets you use default arguments for all arguments left off, including – if there are no arguments at all – running the program as if the command line above had been typed.

```
import sys
if __name__ == '__main__':
    # Read arguments from command line; or use sane defaults for IDE.






    argv_len = len(sys.argv)
    inputFile = sys.argv[1] if argv_len >= 2 else "P0-sample.txt.gz"
    outputFile = sys.argv[2] if argv_len >= 3 else "P0-sampleout.txt"
    k = int(sys.argv[3]) if argv_len >= 4 else 7
```

Grading Rubric

Your P0 submission is graded pass/fail (0/1). It passes if you:

- Submit the code using a zip file with the correct structure with all of the expected files in the right place with the right names
- Provide code that compiles
- Upload code that successfully runs `P0-sample.txt.gz` and `P0-test.txt.gz`
- We will exercise your code with some additional tests, but you pass even if you do not successfully handle those cases.

Remember that you need to pass P0 to get full marks on the rest of the assignments.

-  [P0-sample.txt.gz](#) +
-  [P0-sampleout.txt](#) +
-  [P0-test.txt.gz](#) +
-  [P0java.zip](#) +
-  [P0python.zip](#) +

September 6 2023, 12:48 PM
September 6 2023, 12:48 PM
September 6 2023, 12:48 PM
September 6 2023, 12:48 PM
September 6 2023, 12:48 PM

Submission status

Submission status	This assignment does not require you to submit anything online
Grading status	Not graded
Last modified	-
Submission comments	<div>▶ Comments (0)</div>

 [Contact site support](#) 

You are logged in as [Arjun Senthil](#) ([Log out](#))

Powered by [Moodle](#)