CS2105        **Assignment 0** (3 Marks)        Sem 2 AY24/25

## Submission Deadline

6th Feb 2025 (Thursday) **23:59**. 1 point penalty will be imposed on late submission (Late submission refers to submission or re-submission after the deadline). The submission folder will be closed on 13th Feb 2025 (Thursday) **23:59** and no late submission will be accepted afterwards.

## Objectives

- This is a warm up assignment to familiarize you with programming skills that will be useful for later assignments.

- This programming assignment is worth 3 marks. All the work in this assignment shall be completed **individually**.

## Grading

- We accept submission of Python 3 (3.10.12 in particular), Java, or C/C++ programs.

  - We recommend you to use **Python 3** for your assignments though, as all sample codes or skeleton programs are provided in Python 3.

- Programming languages other than Python 3, Java, and C/C++ are not allowed.

- We will test and grade your programs on the `stu` server. Please make sure that your programs run properly on `stu` and not only on your own system.

- By default, we use the `python3` program installed in folder `/usr/bin` on `stu` for grading. Unless stated otherwise for individual tasks, you are allowed to use libraries installed in public folders of `stu` (e.g. `/usr/lib`) only.

- Your programs will be graded automatically using scripts.

  - Please make sure that your programs behave exactly the same as described in this document, because the grading scripts are unable to award partial marks.

- A set of testing scripts is released to you in the assignment package.

  - These test scripts cover common cases that your programs are expected to handle.

  - During actual grading, we will use extra cases to test your programs. Hence, passing all the released test cases does not guarantee that you will get full marks.

- In addition, <mark>we will detect</mark> fraudulent cases such as hard-coding answers within programs and deduct marks accordingly.

- If you choose to use Java or C/C++, we will compile and run your program for grading using the default compilers on `stu` (`openjdk 21.0.5` installed in `/usr/local/jdk/bin`, or `gcc 11.4.0` installed in `/usr/bin/`).

- The programming language is inferred from the <span style="color:red">extension name</span> of your source code during grading. Therefore, please use extension names to indicate the language of your programs. For example, if a task requires submission of `Checksum.py`, you should provide `Checksum.java`, `Checksum.c`, or `Checksum.cpp`, depending on your programming language.

  - For a Java program, the class name should be consistent with the source file name, and please implement the static `main()` method so that the program can be executed as a standalone process after compilation.

<span style="color:red">We will deduct 1 mark for failure to follow instructions.</span>

## Common Issues

- Incorrect File name causing evaluation script failure: Make sure that your programs have correct names., and avoid using whitespaces in your directory/file names.

- Incorrect output format: Make sure that every line of your output is properly ended with a line break (i.e. "\n" character). In particular, your program should end the output with a line break, unless otherwise stated. Note that the print functions of some languages will automatically add a line break to your provided output.

- Output mismatch: Do not output irrelevant messages that are not shown in sample runs (for example, debugging messages).

## Accessing the `stu` Server

To test your programs, please use your SoC UNIX ID and password to log on to `stu` first:

- If you don't have your SoC UNIX account, please create it here:

  https://mysoc.nus.edu.sg/~newacct

- If you forget your SoC password, please reset it here using your NUSNET ID and password:

  https://mysoc.nus.edu.sg/~myacct/resetpass.cgi

- You will need to be connected to SoC's VPN to access `stu`. The instructions can be found in the link below (you may need to be logged in to view it):

  https://dochub.comp.nus.edu.sg/cf/guides/network/vpn/start

- If you are using a UNIX-like system (e.g. Linux, Mac, or Cygwin on Windows), SSH should be available from the command line and you may simply type:

  `ssh <SoC-UNIX-ID>@stu.comp.nus.edu.sg`

- After logging on to `stu`, you can run a Python 3 script with the following command:

  `python3 <path-to-script>`

- To copy files to `stu` on command line, use `scp`. All `scp` commands should be run **while not ssh-ed** to the `stu` server. The usage is as follows:

  `scp -r <source-folder> <SoC-UNIX-ID>@stu.comp.nus.edu.sg:<target-folder>`

- To copy files back to your system from `stu` on command line, use `scp`. The usage is as follows:

  `scp -r <SoC-UNIX-ID>@stu.comp.nus.edu.sg:<source-folder> <target-folder>`

- If you are using a Windows machine, you may use the above commands with Command Prompt/PowerShell, or install an SSH client (e.g. "PuTTY") if your machine does not have one installed. You may use the "PSCP" or "PSFTP" software (bundled with "PuTTY") to upload your programs to `stu` for testing.

- To connect using PuTTY, enter `stu.comp.nus.edu.sg` as the Host Name. Leave the other settings as is (port number 22, connection type SSH).

- To transfer files via a manner similar to UNIX's `scp` command, you will first need to add the PuTTY folder to your `path` environment variable. This can be done as follows:

  - Using your search bar, search for "path". Click on the first search result (which should prompt you to edit your system's environment variables).
  - Click on "Environment variables..." at the bottom right of the window that appears.
  - Select "Path" at the top of the window that appears, followed by the "Edit..." button.
  - Click on the "New" button to add a new entry to the path, then enter the path to your PuTTY folder. Under the default install directory, this should be `C:\Program Files (x86)\PuTTY`

- Once done, you can use the following command to transfer files to the `stu` server via command prompt `cmd.exe`:

  `pscp -r <source-folder> <SoC-UNIX-ID>@stu.comp.nus.edu.sg:<target-folder>`

- The command may fail the first time due to not finding a target folder with that name on the `stu` server. If this happens, just run the command again (running the command the first time should have created the folder if it does not exist).

- Similar to `scp` above, you can copy files back to your system via:

  `pscp -r <SoC-UNIX-ID>@stu.comp.nus.edu.sg:<source-folder> <target-folder>`

- Otherwise, to use PSFTP, you will need to do the following:

- Run PSFTP (eg. from command line or via Win+R)
- Connect to the `stu` server via:

  `open stu.comp.nus.edu.sg`
- Enter your username, and then your password on the following prompts.
- You should now be connected to the server.

- PSFTP requires you to keep track of 2 different directories: the directory you are in on the server, and the directory you are in on your local machine.

- For the directory on the server:
  - Find out which directory you are currently in via:

    `pwd`
  - Change the directory via:

    `cd <folder-path>`
  - List contents of the directory via:

    `ls`

- For the directory on your local machine:
  - Find out which directory you are currently in via:

    `lpwd`
  - Change the directory via:

    `lcd <folder-path>`
  - List contents of the directory via:

    `!dir`

- To transfer folders:
  - From your local machine to the server, use

    `put -r <source-folder> <target-folder>`
  - From the server to your local machine, use

    `get -r <source-folder> <target-folder>`

- More commands can be found by typing `help` into the prompt on PSFTP, or via the following link:

  https://the.earth.li/~sgtatham/putty/0.52/htmldoc/Chapter6.html

## Testing Your Programs

- Upload your programs along with the `test` folder from the package to `stu`.
- Make sure that your programs and the `test` folder are in the same directory.
- Run `bash test/<Exercise Name>.sh` for testing.
  - For example, to test your program for Exercise 1, run the following command:

    `bash test/Checksum.sh`

- By default, the script runs through all test cases. You can also choose to run a certain test case by specifying the case number in the command:
  ```
  bash test/Checksum.sh 3
  ```

- To stop a test, press and optionally hold `Ctrl-c` if pressing once does not exit the test.

- If you use Java or C/C++, we will compile your program **each time** you run the testing script. The compiled code is stored in a temporary folder and removed after testing, in order to avoid affecting files under your own folders.

- You may assume that all input data to your programs are valid. Hence, there is no need to perform input data validation in your programs.

If you have any question or encounter any problem with the steps above, please post your questions on Piazza or consult the teaching team.

## Program Submission

- Please create a zip file containing your source files (only) and submit it to the `Assignment 0` assignment on Canvas. Source files refer to files containing your own written code (in this case, `Checksum.py` and `PacketExtr.py` (or other file extension, for other supported languages))

- The file name should be `<Matric Number>.zip` where `<Matric Number>` is your matriculation number which starts with letter `A`. An example file name would be `A0165432X.zip`.

- If resubmitting, Canvas may automatically append a number to the end of your submission eg. `A0165432X-1.zip`. This is fine; simply ensure that your file is correctly named before uploading, and allow Canvas to add on the number as necessary. **Do not append the number yourself.**

- All file names, including the zip file and all source files within the zip, are **case-sensitive**.

- In addition, your zip file should not contain any folders or subfolders or irrelevant files such as test.jpg.

- Although we always try to find your actual source files during grading, you are encouraged to submit only once, or otherwise only your latest submission will be graded.

**You are not allowed to post your solutions to any publicly accessible site on the Internet.**

## Policy on AI Tools

**Caution should be used when using AI tools (eg. ChatGPT).** Usage of such tools may be considered plagiarism. Please see NUS' policy on AI for academic works:

In general, the use of AI is acceptable so long as it does <u>not</u> involve code (eg. clarifying concepts taught during lecture, in English or other non-programming languages).

If AI is used for code (generation, or touching up of original code), the relevant sections of code should be marked accordingly with comments, but take note that excessive use of AI for code can be considered plagiarism in the same manner as use of code written by another person, as this assignment is meant to be your own work.

Example comments:

```
#AI Usage: Code generated by ChatGPT using prompt "reading standard input in Python"
(code)
#End section
```

or:

```
#AI Usage: Code touched up by ChatGPT using prompt "reading standard input in Python"
(code)
#End section
```

Any section of code not marked as such shall be considered a declaration that the code did not use AI for generation or touching up in any way.

## Plagiarism Warning

You are free to discuss this assignment with your friends. But, ultimately, you should write your own program. We employ zero-tolerance policy against plagiarism. If a suspicious case is found, student would be asked to explain his/her code to the evaluator in face. Confirmed breach may result in zero mark for the assignment and further disciplinary action from the school.

## Question & Answer

If you have any doubts on this assignment, please post your questions on Piazza. We are not supposed to debug programs for you, and we provide support for programming queries on a best-effort basis only. The intention of Q&A is to help clarify misconceptions and give you necessary directions.

## Exercise 1 – Checksum (1 mark)

Checksum can be used to detect if data is corrupted during network transmission (e.g. a bit flips from 0 to 1). Write a program `Checksum.py` to calculate the CRC-32 checksum for a file <src> entered as command-line argument. File <src> should be placed in the same folder as `Checksum.py`.

You may use the `crc32()` function from the `zlib` library to calculate the CRC-32 checksum (see https://docs.python.org/3/library/zlib.html#zlib.crc32 for details. Remember to import this library before use!). Firstly, you will need to read all the bytes from a file and store them into a `bytes` object. You should open the file with **b**inary **r**eading mode by using **"rb"** as the mode argument of `open()`. Then, you can call the `read()` method of the file object and get the entire file content in a `bytes` object. Now you can get the checksum by directly calling `crc32()` with the file data, and finally print this unsigned 32-bit checksum.

An example code snippet is given below.

```
with open("test.jpg", "rb") as f:
  bytes = f.read()
checksum = zlib.crc32(bytes)
```

Note: in practice, you may encounter an out-of-memory error if you attempt to load a huge file entirely into the memory. In this exercise, however, we limit the maximum file size to 10MB so that you can do so safely without triggering such errors.

If you use Java, you may want to import the `java.util.zip.CRC32` class. If you use C/C++, you can call the `crc32()` function of `zlib` as well, and we will link your program to `zlib` using `-lz` during grading.

**Sample run:**

$ python3 Checksum.py test/test.jpg

3237218320

## Exercise 2 – PacketExtr (2 marks)

In this exercise, you are going to write a program, `PacketExtr.py`, to read consecutive "packets" from the `stdin` stream, extract their data payloads in a *responsive* manner and output to `stdout`.

We define a custom format of packets for this exercise. A packet consists of a text-based header and a binary data payload following the header. The header is a string formatted as "`Size:␣<size>B`", where ␣ represents a white-space character, and `<size>` is a decimal integer representing the number of bytes of the binary data following the header. The header is ended with the "`B`" character, and the payload immediately follows the header without any byte (such as "\n") in between. For example, "`Size:␣2105B`" is a complete and valid header.

The binary payload can contain any byte, not limited to printable characters. Therefore, the payload should not be treated as string data in your program. This packet format clearly defines the boundary between header and binary data within a packet, and also the boundary between consecutive packets. It ensures that all information can be parsed correctly over a data stream.

Packets are fed to your program sequentially through `stdin`, until End-Of-File (EOF) is encountered. It is guaranteed that all packets have correct formats and correct payload sizes. Your program should be **responsive** to the input in the sense that upon receiving a full packet, it outputs the payload of the packet to `stdout` without any extra characters including newline. Unlike Exercise 1, the program cannot read all data once and process them in batch.

It is recommended that your program reads and writes data in binary mode instead of the default text mode. For interactive I/O, `read1()` should be used instead of the ordinary `read()`, and `flush()` should be called after each `write()` to avoid delays caused by buffering. Both `read()` and `read1()` accept one argument which is the maximum number of bytes to read. The main difference is that, `read()` returns only when the specified number of bytes are read or End Of File (EOF) is encountered, while `read1()` returns immediately upon new data stream in and may return fewer bytes than specified. While it is possible to complete this exercise using `read()`, you are **strongly encouraged** to use `read1()`, as this knowledge will be useful in later assignments. You can learn more about the details at https://docs.python.org/3/library/io.html#io.BufferedReader and https://docs.python.org/3/library/io.html#io.BufferedWriter. The following code shows how to do binary I/O on `stdin` and `stdout`:

```python
import sys
# read **at most** 5 bytes from stdin
data = sys.stdin.buffer.read1(5)
# write data to stdout and flush immediately
sys.stdout.buffer.write(data)
sys.stdout.buffer.flush()
```

Here, the `data` object is of `bytes` class. Python 3 programs can operate on binary data using `bytes` objects. This class is very similar to `str`, the string class. For example, both classes have functions `find()`, `split()`, and also slice operators for range access (e.g. `a[0:10]`). Details about this class can be found at https://docs.python.org/3/library/stdtypes.html#bytes-and-bytearray-operations.

The following shows how to manipulate `bytes` objects:

```
# prepend b to the '...' expression to form a bytes object
# instead of str
pos = data.find(b'x')
if pos >= 0:
  # if byte 'x' is found in data
  part1 = data[0:pos+1]  #  this is similar to str slicing
  part2 = data[pos+1:]  # slice until the end of data
```

To convert a `bytes` object to `str`, your program can call the `decode()` method of `bytes`. In this exercise, there is no need to worry about text encoding, as we only use basic ASCII characters in headers.

Finally, to detect EOF on `stdin`, your program can check the length of the `bytes` object read from `stdin`. If your program expects to read more data from `stdin` but receives a zero-length `bytes` object, it means that there is no more data on `stdin` and EOF is encountered.

When testing your program on command line, you can feed the contents of a file to `stdin` of your program using file redirection (<), instead of typing into the terminal. You can also use another type of redirection (>) to save the output of your program to a file rather than let it print to the terminal. For example, the following line feeds the file `input.data` to the program and save its output to `output.data`:

```
python3 PacketExtr.py < input.data > output.data
```

You can then compare binary contents of `output.data` and the given reference output by running:

```
cmp output.data ref-output.data
```

By default, `cmp` outputs all differences found between the two files. Hence, no output means that the two files are identical.

Note that above command line testing can only test your program's correctness. During grading, in addition to testing correctness, we will test responsiveness by setting a **one-second timeout** after feeding your program a full packet. That is, if your program does not output the packet payload on time, we will deem your program as unresponsive. This timeout is sufficient for our test data sizes. This timeout is also present in the provided test script (refer to Page 4 on how to run).

We reiterate that your program should not do batch processing, as interactive processing is one of the key points of this exercise. We limit the size of each packet to 1MB

and the size of all packets to 10MB.

**Sample run:**

```
$ python3 PacketExtr.py < test/cases/PacketExtr/1.in > run-a.out
$ cmp test/cases/PacketExtr/1.out run-a.out
```