Option 1.2: Write a Web Crawler of your own.
Feel free to use any open source crawler as the code base. Write your focused crawler, such as crawling only CoC website or crawling only healthcare web pages. You are expected to crawl at least 1000 pages.

Here are some steps you may follow in a similar way as those in Option 1.1:

(1) Select a seed URL to initialize your Web crawler, such as cc.gatech.edu, and you are expected to crawl at least 1000 URLs.
(2) Show the design of your Web archive to store your crawled web pages, the keywords (subjects) you have extracted.
(3) Plot the crawl speed in terms of the number of keywords you have extracted and the number of URLs you have extracted and the number of URLs you are able to crawl.
(4) Discuss your experience and lessons learned. Predict how long your crawler may need to work in order to crawl 10 millions of pages and 1 billion of pages.

Deliverable:

(a) Source code and executable with readme.
(b) Discuss the design of your crawler: Pros and cons.
(c) Screen Shots of your Web Crawler's Command lines or GUIs
(d) Crawl Statistics (Crawl speedà #pages/minute, ratio of #URL crawled / #URL to be crawled, etc.) in excel plots or tabular format
(e) Discuss your experience and lessons learned.

Design Overview:

The design of the crawler is a recursive approach where the crawler starts from a base URL, visits internal links, and collects the titles of the web pages. The crawler will then collect the word count from each page title, track the keyword word count, and visualize the data with plots. It also stores information in an excel sheet of all websites that it crawls.

Pros:

- The crawler focuses on internal links starting from the base URL, ensuring it remains within the boundaries of the targeted website. This is useful for site-specific crawlers.
- The crawler tracks data (cumulative crawl time, cumulative keyword counts), which provides data on the specific crawl
- It plots in matplotlib the crawl speed and keyword count over time, which can be useful for performance analysis and optimization.
- The crawler handles errors in a basic way, ensuring that failed requests don't halt the entire crawl process.
- It stores information in a excel sheet

Cons:

- The crawler runs sequentially, which may be inefficient for large websites. For better performance, parallel or asynchronous crawling can be implemented.
- Although the crawler catches exceptions, more sophisticated error handling could be implemented
- The crawler isn't designed to handle large-scale web scraping tasks and may face issues if it needs to crawl hundreds of thousands of pages.

Discuss your experience and lessons learned

This project provided a real time opportunity to work with web scraping, data visualization, and data exporting. Through building this project, I learned how to crawl a website, parse HTML content, and extract useful data. It also gave me the chance to experiment with Python libraries like requests, BeautifulSoup, pandas, and matplotlib to manipulate and visualize the data. I found it extremely interesting how it also went through pages in different sections, like main content, or content navigation. One of my favorite parts was finally getting it to work, and exporting valuable content into an excel sheet in a systematic way.

I learned that crawling a large website could take significant time, especially without optimizations like parallelism or multithreading. My program would take over 10 minutes to run completely! I also learned about how to debug and handle certain errors! Overall this project was extremely fun and rewarding.