**Nashr El Auliya**

**Prakruti Dholiya**

**Nourhan El Sherif**

**Yidi Wu**

**Project Documentation**

**GitHub Link:** <https://github.com/nourhanelsherif/EC327_Final_Project_22>

**Introduction**

One of the preliminary steps of an engineering project includes creating a Bill of Materials (BOM), which is a comprehensive list of all the components required to build a product. However, this task can be a very time-consuming process. Traditionally, engineers may spend a long time searching the web for the most optimal product that fits the item requirements and maintains a good quality while remaining within the allocated budget. Thus, our team aimed to create a program to automate a Bill of Materials (BOM) for electronic components specified by the user from the Newark Electronics Catalog. The program takes into consideration the user’s budget and informs the user if the list of items is attainable with the specified budget. Taking the quantity of each item into consideration, the program finds the optimal quantity to purchase in order to get the best deal from Newark. The final output of the program is a csv file of the BOM with the cheapest option of each item. By recommending the best option for each item, the program spares the user from many hours of searching the internet.

**Web Scraping Data and Error Checking Algorithms**

Our program webscrapes the first page of the searched products for all of the possible prices and quantity options available for customers. The program then passes that information to a quantity optimization function (further described in the optimization section). We imported from three main libraries including Beautiful Soup 4, Selenium, and Pandas, which are essential web scraping libraries for Python.

The program first opens Google Chrome using the webdriver. It accesses the Newark website using the base url for the website then uses the function get\_uml to add the user’s inputted search term into the url, which sets the correct page to scrape from. Next, the program searches through the page’s source code for the class “productRow”, which indicates how many items are present on the webpage. Saving this number in variable temp, the program can error check if no parts are found on the webpage. Otherwise, the program iterates through all the items displayed on the web page using a for loop.

The loop moves on to check if the item is currently available in stock on Newark by searching for the class “availability” on the page source code. If an item is found in stock, the program will scrape all of the possible quantity sets that could be purchased, the minimum quantity that could be purchased. The program error checks if the minimum quantity is less than the user quantity. If that passes, then the quantity number is stored in the quantity\_list list and the price corresponding to each quantity in the price\_list list. These two lists are then passed as arguments to the optimized quantity function (OptQuant), which are stored to the final cost list and final quantity list. Thus, the best data values for each search result are stored in the following lists: total\_cost and quantity\_list. In addition, that choice’s website, and part name, are stored in a list as well, all of which will be used by the main algorithm of the program.

**User interface using Graphical User Interface (GUI)**

A GUI is crucial in facilitating the user experience to achieve a more user-friendly interface than the command line. It enables users to The GUI was designed using the tkinter python library. A few main functions were called to create the GUI. The Tk function creates an instance of the tkinter frame. Using the winfo\_children method, we printed all of the child widgets, including part, quantity, and budget. Geometry was called to create the size of the window as 300x100. The Entry function generates all of the input boxes that the user interacts with. All of the input box titles were generated by the Title function, which are essential to prompt the user to input the necessary values. Label prints all of the messages within the GUI boxes, including error messages when the program is unable to find the specified item and when the BOM list exceeds the specified budget, both of which are outputted to a new window. The Pack function finds the ideal position for the texts and input boxes automatically.

We implemented two buttons, Generate and Clear, using the Button function. The first instance of the Generate button generates a new window with the BOM search input list for part name and quantity. If the number of item input was greater than 5, a new column of inputs was generated to accommodate the space necessary to generate all of the input boxes and corresponding titles. We decided to limit the number of entry boxes to generate to a maximum of 10 in order to control the program’s runtime to give the user a better program processing experience. If the user wanted to search more than 10 items, they could simply run the program multiple times. The second instance of the Generate button prompts the program to begin web scraping by calling the web\_scrape function. Finally, the Clear button calls the CleanUp function on the current window to remove all of the user inputs, displaying blank entry boxes.

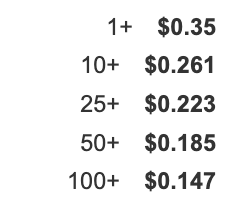
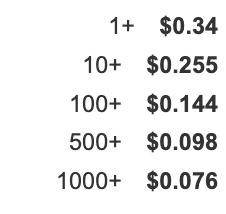
**Simplifying Data Using Optimize Quantity Function**

After web scraping all the relevant information, the price optimization function is called to find the optimal quantity to purchase within the range of the user’s input quantity. This is because the Newark website offers a unit discount for purchasing larger quantities of an item. Thus, this price optimization function simplifies the data that the remainder of the program must sort through to find the cheapest option between all variants. This function was key for increasing the program's efficiency and processing speed significantly.

The initial output of the data scraped has multiple prices per unit (PPU) that decrease as quantity increases:

Example:

*Choice 1:* *Choice 2:*

Function to simplify the scraped data takes in the list of minimum quantities, price per unit, and the quantity the user wants. This function will calculate the difference between the user’s quantity and quantity options, iterate through the difference and find the lower and upper quantity options. It will then calculate if it is cheaper to buy the upper bound quantity or buy the lower bound quantity. The output will be the best price and the quantity the user should buy.

Ex. If the user wants 24 units of the above option, this would calculate the price of buying 24 units at the PPU of 10+, and compare it to the price of buying 25 units at the PPU of 25+. In this instance, it is cheaper for the user to buy 25 units at $0.223/unit than 24 at $0.261/unit, and therefore the function will tell the user to buy 25 units with a total cost for this item as $5.58. This function call would then be repeated for Choice 2. Later on, the two options will be compared and the cheapest of the two will be chosen.

This function is called for each option that was scraped, and thus simplifies the above outputs to just one option per result choice.

Example, the above results will simplify to:

*Choice 1:*  *Choice 2:*

Buy 25 units, $5.58 Buy 24 units, $6.12

**Algorithm**

The algorithm that sorts through the available variants per part and locates the cheapest/most cost-efficient choice is the generate() function. The generate() function–as the name suggests–generates the output .xlsx BOM file. It also produces the GUI window that outputs the total cost, and whether the final BOM is over/under-budget.

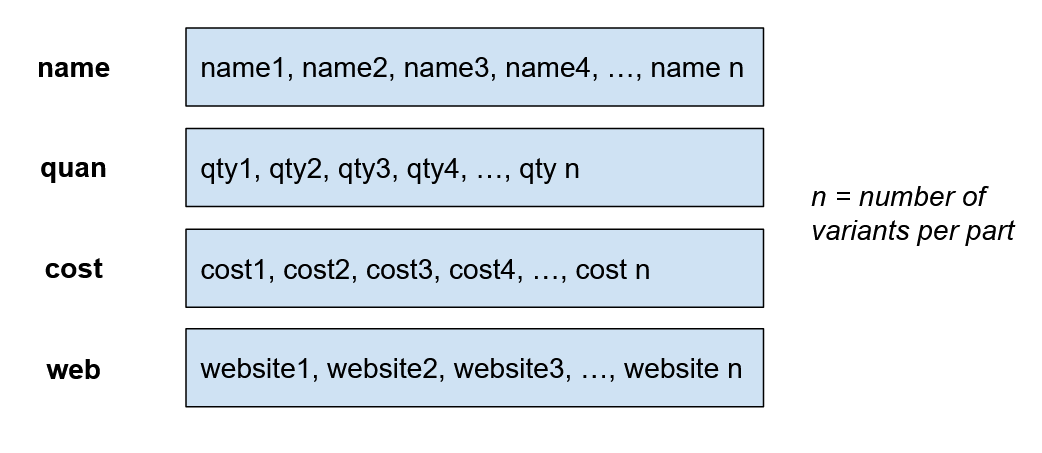
The function uses a for-loop to call our web-scraping function n amount of times, where n is the number of parts that the user has determined to be in their BOM. Each time the web-scraping function is called, it returns the cheapest option for a given part, then for each iteration of the for-loop it appends the item returned to a list called output\_data. The web scraping function itself returns a list containing the name, quantity, total cost with respect to quantity, and URL of the cheapest option. Using the openpyxl library, the contents of output\_data is translated into an output excel file. generate() then produces a new GUI window that notifies the user that it has generated the BOM, what the total cost of the bill is, and whether the given bill is over/under the inputted budget.

Ignoring our intricate GUI architecture, the heart of the algorithm is the generate() function and the webScraping() function. A GUI that prompts the user for (1) the number of parts in the bill, (2) the budget, (3) and part name as well as quantity per part, which initiates the flow of our algorithm. User interaction (pressing the “generate” button) calls the generate() function to produce a BOM. However, generate() in turn calls webScraping() to get the relevant data it uses and the process described previously takes place. The webScraping() calls OptQuant() to narrow down the data being collected. This is then sifted to produce the BOM output.

**Data Structure (how data was stored)**

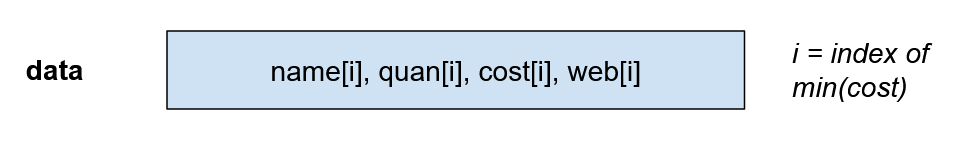
All of the significant data in our program is stored in python’s list data structure. However, where they are instantiated, returned, or nested is what affects the way our program operates. First, we have the lists that store all the variations of a given part in webScraping(). Then we have a singular list that webScraping() returns, containing the name, price, quantity, and URL of the cheapest option for a given part. Finally, we have the output\_data which would be translated into the excel file. output\_data is a list of lists, it’s a list of the cheapest options which is in and of itself a list (containing the option’s attributes).

The way webScraping() stores all the information it collected for the variants of a given part type is by storing the relevant information in four different lists: name, quan, cost, and web. Each list contains the name, quantity, cost, and URL of all the variants–respectively, and as such all four lists have the same number of indices. This fact is then leveraged in finding the cheapest option, and consequently in producing the list that webScraping() returns.



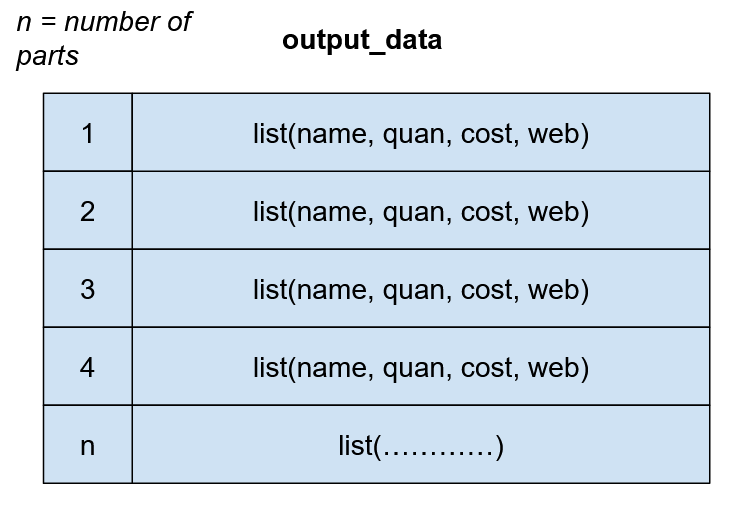
*Figure 1: The containers for all the variants*

We’ve previously mentioned that webScraping() returns a list, this list contains all the attributes of the cheapest option for a given part. By leveraging the fact that name, quan, cost, and web all have the same length, we can simply use python’s given functions to find the index of the lowest value in the cost list and use that index to index into the other three lists. The values returned from the four lists at that specific index i is then stored in the list that webScraping() returns.



*Figure 2: The list that webScraping() returns*

Then in generate(), we have the list output\_data. Using a for-loop, the value returned from each call of webScraping() is appended to this list. As such, output\_data is a list of lists. This is the final container for the data that our program uses as the BOM.



*Figure 3: The final BoM list*

**Displaying output**

Excel and output file

**Limitations**

Our program exhibited some limitations due to some constraints we implemented within our program. These limitations include only allowing 10 user inputs per program run. As explained previously, this allowed our program to ensure a fast runtime, especially since it was created using Python, which is typically considered slower than other programming languages, such as C++. For the same reason, our program only scrapes the first page of the search results list. Another limitation of our Auto BOM is that it currently only scrapes Newark Electronics. This is a huge area for future improvement. By expanding the number of websites that the Auto BOM can scrape, we allow the user to further expand the scope of their engineering project in terms of the required components. Currently, the user is limited to searching for electric components. Additionally, our BOM requires a specific product name that matches the Newark catalog product name. This is necessary because having a specific product name is what allows for proper search results to show up that can be scraped. Otherwise, if the search is not specific enough, the web page will not have a specific search results list that can be scraped because it redirects to a more general Newark Electronics page.

**Future Outlook**

In the future, we can scrape more websites, as well as include a quality component to our algorithm by scraping ratings to add complexity and functionality to the program.

In addition, more functionality can be added to the BOM by asking the user for the material’s significance. The user should input the items by most to least important. Then the program can generate multiple BOMs, with the first being the absolute cheapest BOM, the second being the second cheapest option, with a more expensive first item (which is likely of better quality), and so on. With this, the user can choose between multiple BOMs that are all within budget, and have more money be spent on the more important components.

# numpy data structure

import numpy as np

num\_items = 10 #num of items user wants to search

np\_cost = np.zeros((num\_items, 1))

np\_name = np.zeros((num\_items, 1))

np\_qty = np.zeros((num\_items, 1))

for i in total\_cost:

np\_cost