Administrative:

Team Name: F0rtn1t€_L<3vr$

Team Members (Github): Sophie Shah (sophieshah) and Kristen Moy (kristenmoy)

Github Link: https://github.com/sophieshah/DSA-Project-3

Youtube Link: https://youtu.be/w40ClTiUHGM

Proposal:

The problem we are trying to solve is that it is sometimes difficult to find a movie. With the app Letterboxd, you can search by genre, release date, or any other constraint specified to search for a movie. However, when it lists out the movies based on the constraint, it does not rank them in the highest order of ratings. Letterboxd has honest reviews and ratings by the general public, which is why we find this app so entertaining and, in a way, trustworthy. Simply, Letterboxd does not find movies based on a constraint and rank them by ascending order by ranking. Since this is a problem, it is hard to find a movie that would satisfy my specific wants when finding a good movie. I am usually compelled to watch a movie that is most liked by the general public, which is why this could be useful.

The features that we implemented were different ways of searching through a data set. Based on the constraints that a user inputted, we searched through the data set based on those constraints. For example, if a user wanted to watch a movie in the comedy genre, a list of the top five movies, by rating, in that genre would print out.

Our data set consisted of multiple csv file sheets, such as actors, countries, crew, genre, languages, movies, releases, studios, and themes. We specifically used the files which contained the movies and genres. In the movies csv file, it also contained information about a movie's tagline, description, duration and rating. We were able to search through these two files to find the information needed.

We did not use any tools, or APIs because we did not create this with a web interface as we intended since we became short staffed. We decided that we would run our code through a command line interface, and used libraries to create and implement our data. The libraries we used were iostream, fstream, vector, sstream, map, and algorithm.

The data structures that we used were map and unordered_map. We used the STL map, since we ended up with two members in our group rather than three like we had intended. We implemented the undordered_map from scratch using a 2D vector, with the first index being the key and everything else being the values.

Our distribution of roles were a bit different than how we planned before. At first we were going to equally distribute the workload with all of us implementing each area, such as the frontend and backend code of the interface and map implementations. However, since we lost one of our members, we had to cut down on some of our initial ideas and our web interface because none of us knew React or JavaScript. We both were evenly distributed in implementing the code, but with a smaller workload because of the adjustment. We drew out a game plan and both coded an equal amount as well as the report.

Analysis

We made quite a few changes after the initial proposal. We decided that we were going to a standard command line interface, since we had less people to work on this project. Additionally, instead of searching through the description

Time Complexities

- **insert()**
    - This function replicates an insert function in an unordered map with a string key and a vector of string value. We had to check if our replica unordered_map had duplicate keys. This meant that we had to traverse through our entire vector to check if the key had already been inserted, making our time complexity O(n), where n is the number of keys in the replica map, or more specifically the rows in the 2D vector.
- **begin()**
    - In this function, we replicated the begin function in an STL unordered_map. Since the vector has a built-in function, the time complexity of this function would be O(1). Finding the beginning of the vector replica map is a constant function.
- **end()**
    - Similar to the begin function, it is a replica of an STL unordered_map. Implementing this from scratch using a vector, it has a built in function already. The time complexity of this function is O(1), since this vector replica map's built in function is in constant time.
- **find()**
    - In this find function, we had to iterate through every "key" in the 2D vector replica of the unordered_map to see if the inputted key was in the map. This called for iterating throughout all of the keys in the vector – specifically through the number of rows in the vector. The time complexity of this function would be O(n), where n is the number of "keys" in the replica unordered_map, or more specifically the rows in the 2D vector.
- **Entire Movies class**
    - The Movies class which consisted of a Movies constructor, getId(), getName(), getRating(), getDate(), getDuration(), and getDescription() all had a time complexity of O(1) because they were all return functions, or intializations. These were all in constant time, thus being O(1).
- **findBestMovieUnordered()**
    - This function has multiple loops. The outer section of the first nested loop iterates through the entire unordered map um, having a complexity of O(n) where n is the number of elements or movies in the map. Inside of this loop, there is another one that iterates through all the genres stored in the given vector controlled by the outer loop. This iterates O(g) where g is the number of genres a given movie contains. That means this first loop is O(n * g). Then the function uses std::sort

which has a complexity of O(mlogm) where m is the number of elements in the topFive vector, or the number of movies that are in the given genre. Then, there is a loop that iterates through the topFive vector five times, which is constant time, or O(1). The total time is O((n * g) + mlogm), where n is the number of elements/movies in the map, g is the number of genres held with a given movie, and m is the number of movies in the map that contain the given genre.

- **readMovies()**
    - This function reads through the csv file to get the movies. The getline function has a time complexity of O(n), where n is the length of the string being read. We have a for loop that reads each line of the csv file for each unique movie in the csv file. Every time readMovies() is called, the for loop will iterate 100,000 times each time. This for loop calls getline to get the movieId, name, rating, date, minute, and description 100,000 times. So, the time complexity would be O(100,000*n) => since this for loop runs 100,000 times, which is in constant time, we can drop the constant values. This simplifies to O(n).
- **makeUnorderedMap()**
    - In this function, we created the unordered_map replica. We iterated through movieVect, which had a constant size of 100,000 based on readMovies. Additionally, we iterate through the genre csv file 100,000 times and used getline functions, to read the genres. These have a time complexity of O(n), where n is the length of the string. Overall, this function would have a time complexity of O(100,000*n). Since the for loops will iterate 100,000 time each time, this is constant and can be dropped. This function's time complexity ends up being O(n).
- **findBestMovieMap()**
    - This function has multiple loops, similar to the findBestMovieUnordered function. The outer section of the first nested loop iterates through the entire unordered map um, having a complexity of O(n) where n is the number of elements or movies in the map. Within the loop, there is a try-catch block, which itself has constant time of O(1). Also inside of this loop, there is another one that iterates through all the genres stored in the given vector controlled by the outer loop. This iterates O(g) where g is the number of genres a given movie contains. That means this first loop is O(n * g). Then the function uses std::sort which has a complexity of O(mlogm) where m is the number of elements in the topFive vector, or the number of movies that are in the given genre. Then, there is a loop that iterates through the topFive vector five times, which is constant time, or O(1). The total time is O((n * g) + mlogm), where n is the number of elements/movies in the map, g is the number of genres held with a given movie, and m is the number of movies in the map that contain the given genre.
- **makeMap()**

- Similar to makeUnorderedMap, we read through 100,000 lines of the movie and genres csv file. Within each iteration of reading the lines, we use the getline function. The getline function has a time complexity of O(n), where n is the length of the string. This gives us a time complexity of O(100,000n + 100,000n). Since the for loops to read through each line will always run 100,000 times. It is constant. We can drop the constants from the time complexity => O(n + n) = > O(2n). We can further simplify this to be O(n) since we can drop the constants.

- **main()**
  - In our main function, we call the makeMap, readMovies, and makeUnorderedMap, findBestMovieUnordered, and findBestMovieMap. As stated above makeMap had a time complexity of O(l), where l is the length of the string being read. readMovies had a time complexity of O(l) as well, where l is the length of the string being read. makeUnorderedMap had a time complexity of O(l), where l is the length of the string being read. findBestMovieUnordered has a time complexity of O((n * g) + mlogm), where n is the number of elements/movies in the map, g is the number of genres held with a given movie, and m is the number of movies in the map that contain the given genre. findBestMovieMap had a time complexity of O((n * g) + mlogm), where n is the number of elements/movies in the map, g is the number of genres held with a given movie, and m is the number of movies in the map that contain the given genre. To add, we have for loops that print out the top five movies, which is in constant time of O(1). This also pertains to the other output functions as well/
  - Adding all of the time complexities together, we have O(l + l + l + n*g + mlogm + n*g + mlogm). We can condense this into O(3l + 2n*g + 2mlogm) by combining like terms. We can drop the constant values and get our final time complexity as => O(l + n*g + mlogm).

Time Analysis of Two Data Structures:

We used chrono to test the execution time of our two data structures. We started measuring the time when the functions were called, and stopped it when the functions were done. As our parameters, we used “Comedy” as our genre, “2023” for the year, and “3” for the hours. The replica unordered_map that was implemented using a 2D vector instead, has a time of about 532,244 microseconds. The STL map took about 43,580 microseconds to execute. Through this observation, we saw that the STL map took less time to execute.

Reflection

Coding in a group was very different from working individually. Expressing our ideas in a way that both of us could understand was a little difficult as well. The workflow was intimidating at first because we did not know where to start, but as we kept working on it, it became easier. The biggest challenge was adjusting to the change with one of our team members dropping the class. Our planned workflow at the beginning was disrupted, so we were not able to

implement the user interface as we planned, as well as making our program more complex as stated before.

If we were able to restart this project again, we should be more prepared for change and cutting things out of our project. We had plans that would have worked better if we still had three people on our team. Since we were both pretty busy with upcoming exams, our workload got pretty heavy on top of this project. If we were prepared for this change and started the project early, we might have been able to code the interface and other search constraints as we planned.

Kristen: I learned a lot through working on this project because we had the freedom to do whatever we wanted to with the data set. I had previously worked together with other people in labs, but there were certain rules and an end goal that was clear. In working on this project, I had a lot of ideas, but figuring out how to implement them was a little tricky. I learned that projects can be a little intimidating at first, especially when you have full control. Now I know how to approach the situation of creating a project, and I feel less intimidated.

Sophie: I learned through working on this project. Since we had to read through a csv file, I learned more in depth about implementing try and catch statements. I had never done this before, so now I can use this in my future projects. Similar to what was stated before with starting the projects early, I wish we could have implemented everything that we planned because that would have been pretty cool.

References:

Data set: https://www.kaggle.com/datasets/gsimonx37/letterboxd/data?select=movies.csv

Exceptions: https://www.w3schools.com/cpp/cpp_exceptions.asp

Sorting vector of pairs:
https://www.geeksforgeeks.org/sort-vector-of-pairs-in-ascending-order-in-c/

Measuring execution time:
https://www.geeksforgeeks.org/measure-execution-time-function-cpp/