Spell Checker and Movie Rating Analytics

CS 262 - Fall 2024

In this project you will be creating programs that involve inputting and outputting text. Be sure to follow each of the steps below carefully. If you have any questions about the requirements, please ask.

Partners:

You may optionally choose to work with a partner on this project. If you do, only one of you should submit the project to Canvas, but clearly indicate the partnership in your JavaDoc and README.

If you choose to work with a partner, I strongly recommend that you are both always present when your project is being worked on (both of you are present while any code is planned or typed), and that you take turns typing. ***For a successful partnership***, once the project is completed, ***both*** partners should feel comfortable that they could do the project again ***individually*** if asked.

Exercises:

For each exercise below (except for #1), your task is not only to implement a solution that works, but to *design* your solution to be *as **efficient** as possible* based heavily on your use of data structure(s) (collection(s)) and your algorithm. You are ***encouraged*** to use built-in `sort` and search methods (e.g., `binarySearch`) where necessary (remember, what state does an array have to be in for binary search to be applied?). Note that you may want to use more than one data structure for different parts of each single "exercise" below.

As part of your project submission, in your README (see details below), you are expected to explain and justify your data structure choices, so

1. Create a class called **`MyWord`** that:
    a. stores only one thing: a character array (`char[]`).
    b. implements the `Comparable` interface.
        i. When you are comparing two `MyWord`s, ***you are not allowed to use any of the built-in String methods***.
        ii. When sorted, `MyWord`s should appear in reverse alphabetical order. For instance, dog comes before cat. Capitalization does not matter. dog should come before Cat, and Dog should come before cat.

        iii. Finally, remember that equality and order comparison should be in sync. That is, your **MyWord** class should also implement what's necessary so that when two **MyWord**s are compared for equality, the result is in sync with the result from being compared for order.

2. Create a class called **MySpellChecker** and implement a "spell checker" that uses a given dictionary to identify misspelled words and for each ask if the user wants to correct it, ignore it, or add it to the dictionary.

    a. a `main` method that is expected to take two command-line arguments:

        i. a dictionary file location

        ii. an input file location

    If the user doesn't provide these arguments, print a message to tell the user what proper usage of the program should be and end the program.

    b. The dictionary file should contain words separated by whitespace (i.e., a space, a tab or a newline character). Assume the words in the dictionary file have no meaningful order (that is, the words in the dictionary file may be in any order). Read in all of the words in the file and store them (as **MyWord**s). (Note: `String`s have a method `.toCharArray()` that you may find useful.) (Choose how you want to store them to be as efficient as possible for the spell checking task.)

    c. If the input file location starts with "http://", then the input file is located on the web instead of on your computer.

    d. Each word in the given input file should be checked to see if it is in the given dictionary. With the exception of dashes (-) and apostrophes ('), ensure that punctuation marks and other characters that are not letters are ignored prior to spell-checking (Note: `Character` class has a method `.isLetter()` that you may find useful.) If the word is not in the dictionary, show the user the line of text that contains the misspelled word, and tell them the word that is misspelled, like so:

```
Line 1: Humpty Dumpty sat on a wall
 check: Humpty
```

    Then, ask if the user would like to:

        i. 1 - correct it (and if so, keyboard prompt them to provide a correction to replace the misspelled word)

   ii. 2 - ignore it (leave it as is and continue spell checking)
   iii. 3 - ignore all (leave it as is and any future occurrences should be ignored)
   iv. 4 - add to dictionary (the misspelled word should now be considered part of the dictionary; any future occurrences should not be found as misspelled)

e. Prior to the program ending, if any words were "added to the dictionary", the dictionary file that was given should be updated to reflect the addition(s).

f. Finally, if the input file is local (i.e., not on the web), then the spell-checked final message should be written back to the file.

   For example, using the given dictionary file and the following local input file:

```
Humpty Dumpty sat on a wall,
Humpty Dumpty had a grat fall;
All the king's horses
And all the king's men
Couldn't put Humpty together again.
```

   The program output should be:

```
Line 1: Humpty Dumpty sat on a wall,
 check: Humpty
  1 - Provide correction
  2 - Ignore
  3 - Ignore All
  4 - Add to dictionary
>#3

Line 1: Humpty Dumpty sat on a wall,
 check: Dumpty
  1 - Provide correction
  2 - Ignore
  3 - Ignore All
  4 - Add to dictionary
>#3

Line 2: Humpty Dumpty had a grat fall;
 check: grat
  1 - Provide correction
  2 - Ignore
  3 - Ignore All
  4 - Add to dictionary
>#1
Replace grat with: great

Line 3: All the king’s horses
 check: king’s
  1 - Provide correction
  2 - Ignore
  3 - Ignore All
  4 - Add to dictionary
>#3

Line 5: Couldn’t put Humpty together again.
 check: Couldn’t
  1 - Provide correction
  2 - Ignore
  3 - Ignore All
  4 - Add to dictionary
>#4

Final text:

Humpty Dumpty sat on a wall,
Humpty Dumpty had a great fall;
All the king’s horses
And all the king’s men
Couldn’t put Humpty together again.
```

(And the local file should now have the final spell-checked text, and “Couldn’t” should be added to the dictionary file, so running again with the same input files would no longer flag “Couldn’t” as a word to check.)

g. Choose data structure(s)/collection(s) so that your solution is as efficient as possible. In your README file, describe the collection(s) you chose to use for your spell-checker solution and why. If you considered alternatives, describe how you arrived at your decision.

3. MovieLens (https://movielens.org/) is a movie recommendation website created and used for (academic) recommendation research (by GroupLens Research Lab at the University of Minnesota). On the site, users can rate movies in order to receive personalized

recommendations. The dataset of movies and ratings is publicly available for download. Let's use the MovieLens dataset to do some rating analytics.

We will use the 32 million movie ratings dataset (MovieLens 32M ml-32m.zip), and the "small" 100,000 rating dataset (ml-latest-small.zip under MovieLens Latest Datasets under recommended for education and development). Download both of these datasets and store them in your project 3 folder. (Use the "small" dataset while you develop your program/algorithm and choose data structures that will perform most efficiently, but realize that once completed the program will need to work for the much larger dataset and run in a reasonable amount of time.)

We will be using the **movies.csv** and the **ratings.csv** files from each dataset. Read the README files for the datasets and examine the files enough to understand what's in them and how to work with them.

Write a movie rating analytics program with the following requirements:

a) the `main` method should have an optional command-line argument to indicate whether the user wants to use the small dataset or the large dataset: if the argument is present and if it is the String "large", then the large dataset should be used; otherwise the small dataset should be used. (That is, the small dataset should be the default, and the user does not have to enter a command-line argument to use the small dataset, but must add "large" when running the program in order to use the large dataset.)

b) Read in and store the relevant data from the **movies.csv** and **ratings.csv** files. (Choose and use data structures that will provide the best performance and that will support the functionality needed for the program (described below).)

    i) Note that csv stands for "comma-separated values", which means that the values in each line of data are separated by commas. However, some movie titles have commas in them, like I, Tonya, for example. When a movie title has a comma in it, the title portion of the line of data is surrounded by double quotes, like so:

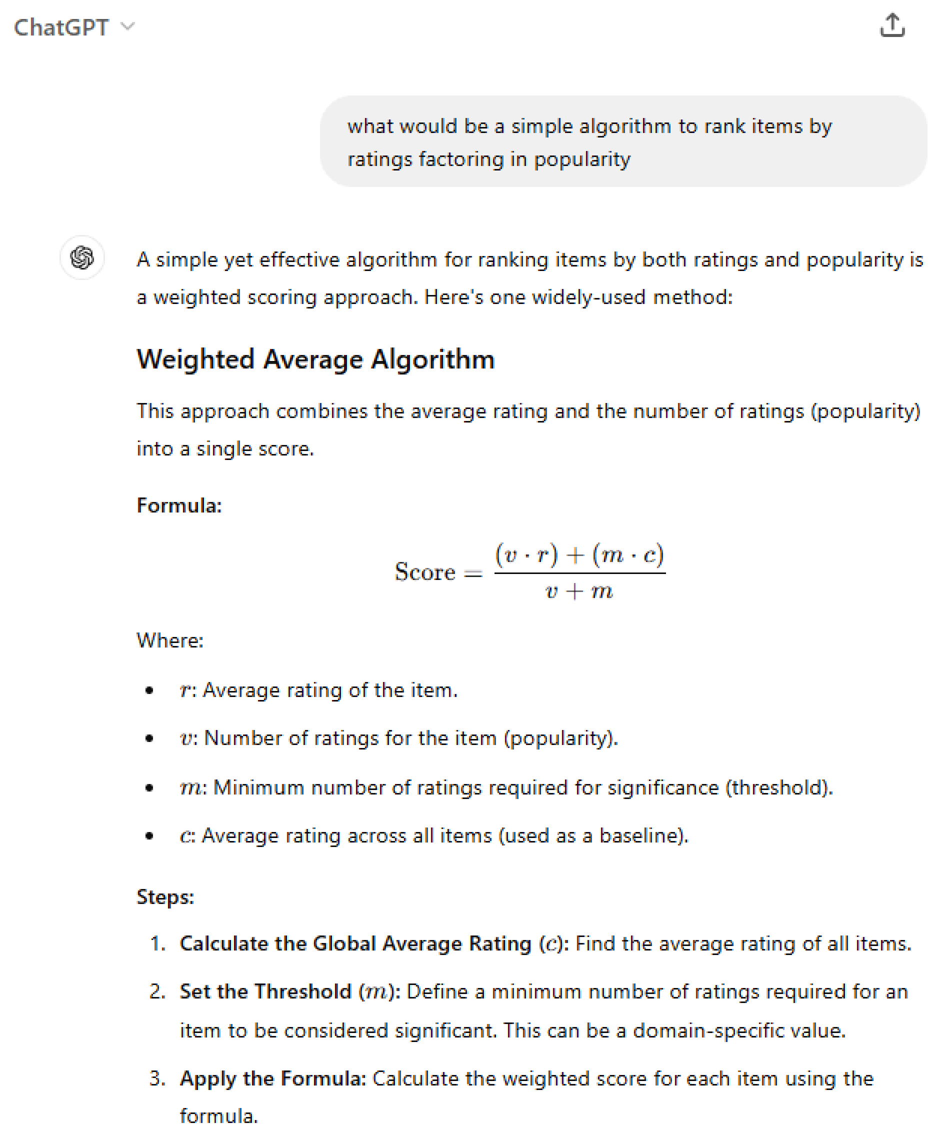
    178061,"I, Tonya (2017)",Drama

    ii) Further, some movie titles have quotation marks in them, like "Great Performances" Cats (1998). The MovieLens dataset "escapes" quotation marks with an additional quotation mark, so in the dataset the Cats example is stored as: ""Great Performances"" Cats (1998). (This movie only appears in the large dataset and the full line in the data file is:

    51372,"""Great Performances"" Cats (1998)",Musical

iii) Finally, note that you could add lines to the dataset files simply to test specific cases. Alternatively, you could make your own separate dataset files to test specific things.

c) To rank Movies, the pure average would be given by the sum of the ratings divided by the total number of ratings. However, this does not adjust for movies that have very few ratings, with which the few ratings are weighted very heavily (e.g., should a movie that is rated 5 stars by only one person truly be taken as a 5-star movie?). As such, we should account for movie popularity in the calculation of a movie's average rating by computing a "weighted average." I asked "a friend" for an algorithm we can use to compute a weighted average for each movie (given its ratings and an average rating across all movies, using 15 as the number of ratings required for significance threshold (m)):



what would be a simple algorithm to rank items by ratings factoring in popularity

A simple yet effective algorithm for ranking items by both ratings and popularity is a weighted scoring approach. Here's one widely-used method:

Weighted Average Algorithm

This approach combines the average rating and the number of ratings (popularity) into a single score.

Formula:

$$\text{Score} = \frac{(v \cdot r) + (m \cdot c)}{v + m}$$

Where:

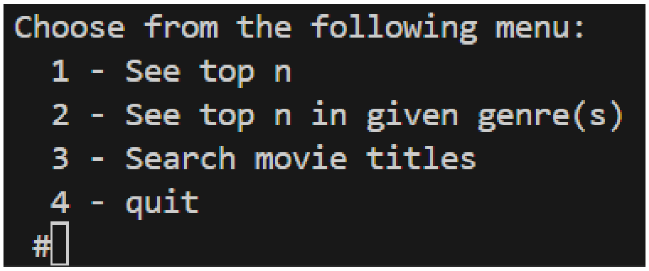
- r: Average rating of the item.
- v: Number of ratings for the item (popularity).
- m: Minimum number of ratings required for significance (threshold).
- c: Average rating across all items (used as a baseline).

Steps:

1. **Calculate the Global Average Rating (c):** Find the average rating of all items.
2. **Set the Threshold (m):** Define a minimum number of ratings required for an item to be considered significant. This can be a domain-specific value.
3. **Apply the Formula:** Calculate the weighted score for each item using the formula.

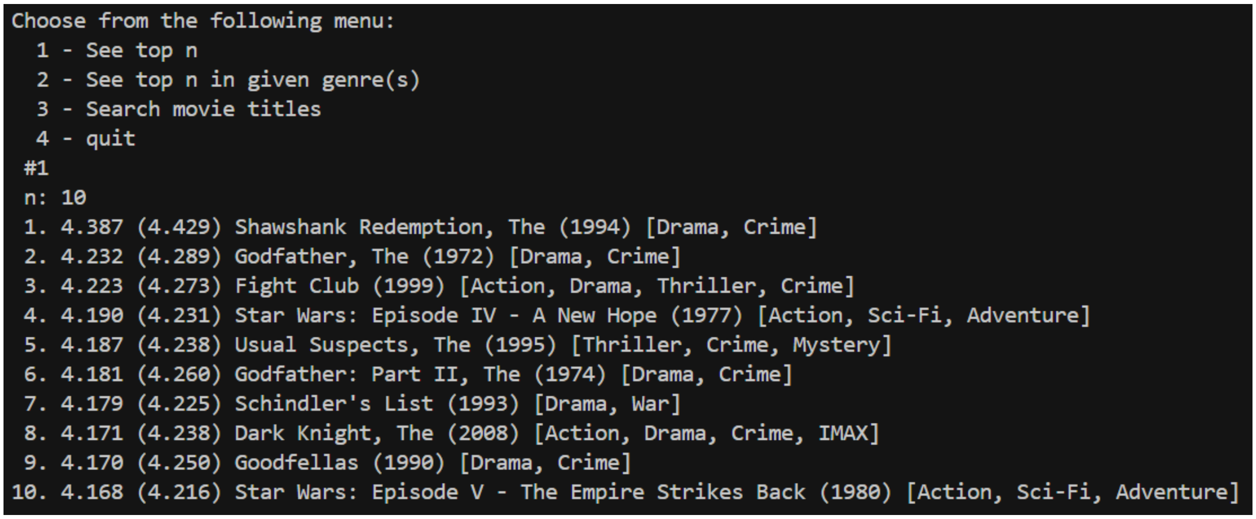
i) Spoiler (But Important Efficiency) Note: (Re-)Computing averages (*r* and *c*) every time you want a weighted average will be very inefficient. You may want to consider an object-oriented approach to storing movies, with instance variables (and class variables) storing running totals (e.g., rating sums), etc.

d) Users should be given the following options:



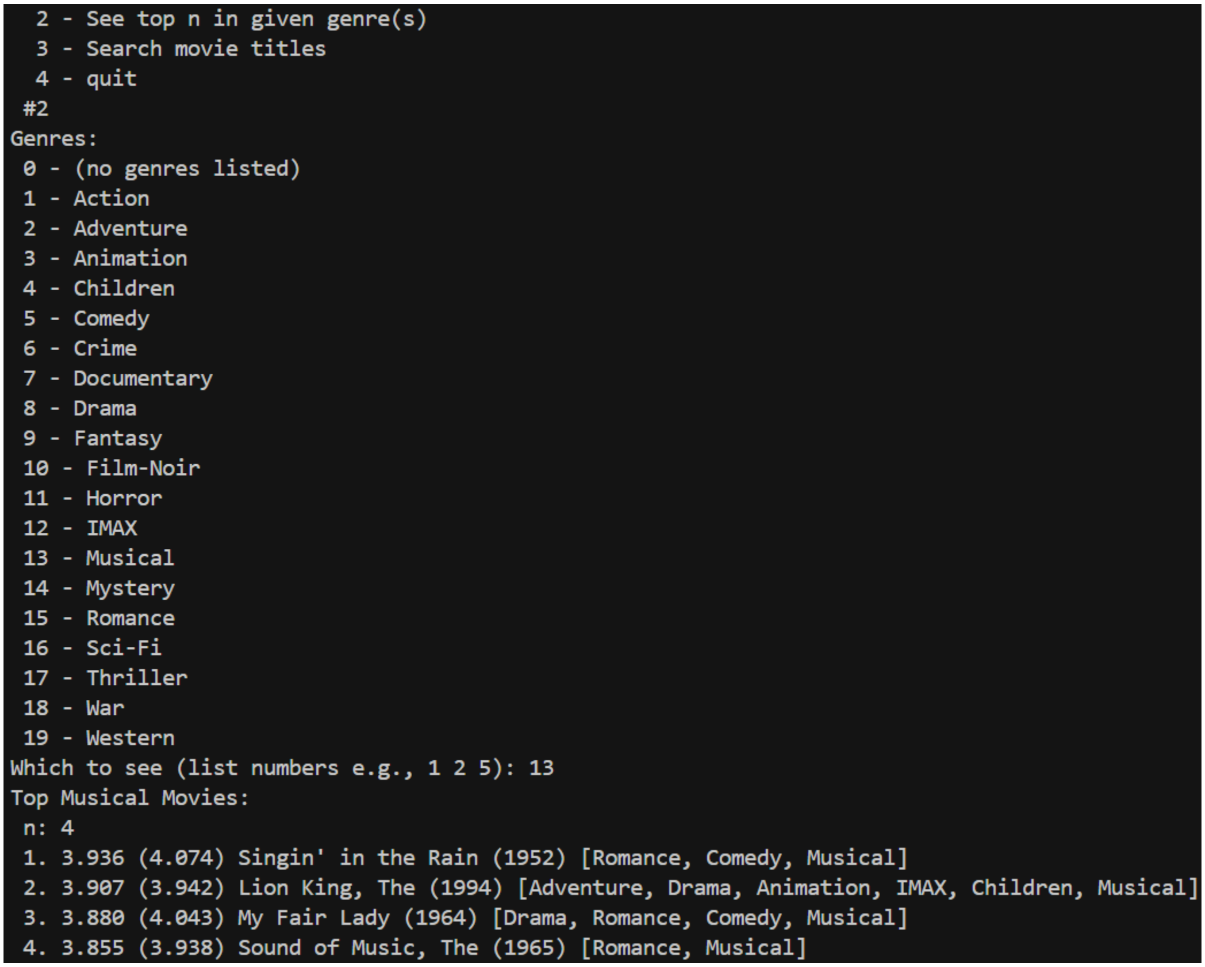
```
Choose from the following menu:
  1 - See top n
  2 - See top n in given genre(s)
  3 - Search movie titles
  4 - quit
 #
```

e) If option 1 is chosen, the user is prompted for an n value and then the top n Movies are displayed, ordered in descending order by weighted average (with the true average in parentheses) displaying the title with its year (same as it appears in the movies file) followed by its genres. For example, using the small dataset, the top 10 movies are:



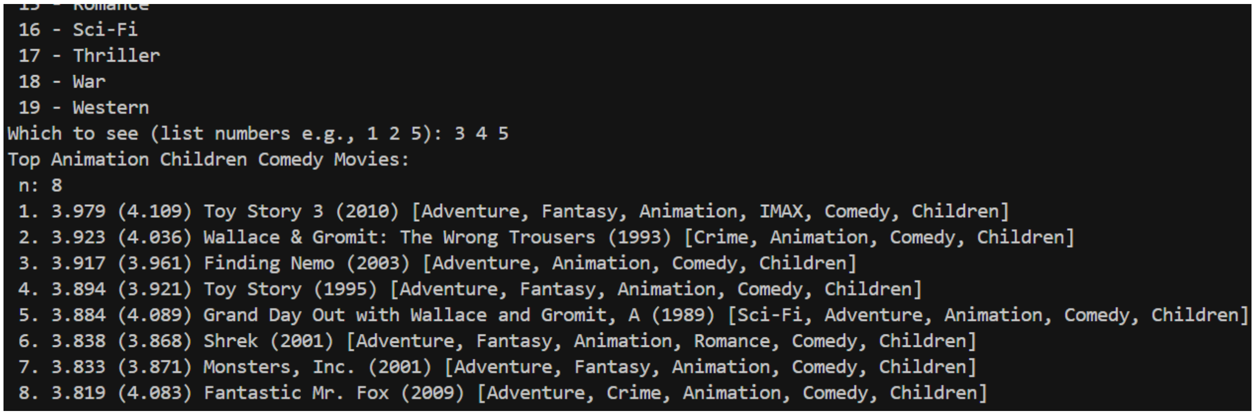
```
Choose from the following menu:
  1 - See top n
  2 - See top n in given genre(s)
  3 - Search movie titles
  4 - quit
 #1
 n: 10
 1. 4.387 (4.429) Shawshank Redemption, The (1994) [Drama, Crime]
 2. 4.232 (4.289) Godfather, The (1972) [Drama, Crime]
 3. 4.223 (4.273) Fight Club (1999) [Action, Drama, Thriller, Crime]
 4. 4.190 (4.231) Star Wars: Episode IV - A New Hope (1977) [Action, Sci-Fi, Adventure]
 5. 4.187 (4.238) Usual Suspects, The (1995) [Thriller, Crime, Mystery]
 6. 4.181 (4.260) Godfather: Part II, The (1974) [Drama, Crime]
 7. 4.179 (4.225) Schindler's List (1993) [Drama, War]
 8. 4.171 (4.238) Dark Knight, The (2008) [Action, Drama, Crime, IMAX]
 9. 4.170 (4.250) Goodfellas (1990) [Drama, Crime]
10. 4.168 (4.216) Star Wars: Episode V - The Empire Strikes Back (1980) [Action, Sci-Fi, Adventure]
```

f) If option 2 is chosen, the user is shown the movie genres to choose from, and is prompted for which genres and how many of that genre *combination* to display. That is, the user may enter one or more genre: if one genre is entered, the user should see the top movies in that genre. If more than one is entered, the user should see the top movies with that combination of genres. For example:



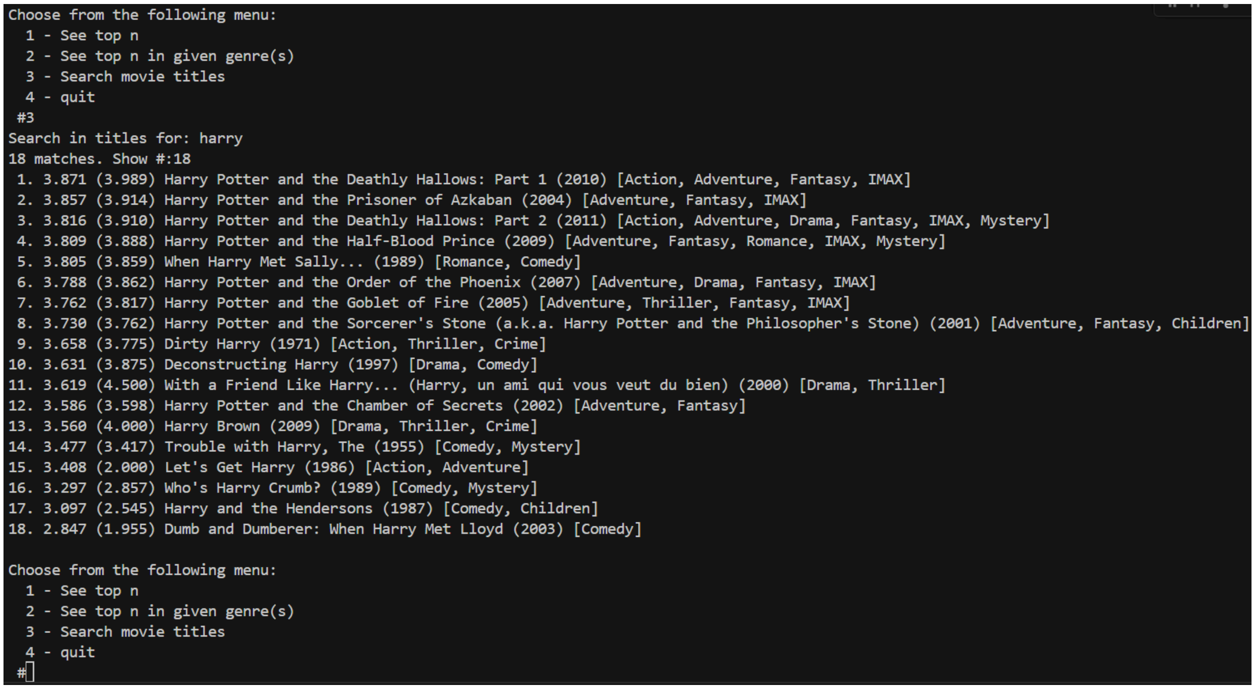
```
  2 - See top n in given genre(s)
  3 - Search movie titles
  4 - quit
 #2
Genres:
 0 - (no genres listed)
 1 - Action
 2 - Adventure
 3 - Animation
 4 - Children
 5 - Comedy
 6 - Crime
 7 - Documentary
 8 - Drama
 9 - Fantasy
 10 - Film-Noir
 11 - Horror
 12 - IMAX
 13 - Musical
 14 - Mystery
 15 - Romance
 16 - Sci-Fi
 17 - Thriller
 18 - War
 19 - Western
Which to see (list numbers e.g., 1 2 5): 13
Top Musical Movies:
 n: 4
 1. 3.936 (4.074) Singin' in the Rain (1952) [Romance, Comedy, Musical]
 2. 3.907 (3.942) Lion King, The (1994) [Adventure, Drama, Animation, IMAX, Children, Musical]
 3. 3.880 (4.043) My Fair Lady (1964) [Drama, Romance, Comedy, Musical]
 4. 3.855 (3.938) Sound of Music, The (1965) [Romance, Musical]
```

…



```
 16 - Sci-Fi
 17 - Thriller
 18 - War
 19 - Western
Which to see (list numbers e.g., 1 2 5): 3 4 5
Top Animation Children Comedy Movies:
 n: 8
 1. 3.979 (4.109) Toy Story 3 (2010) [Adventure, Fantasy, Animation, IMAX, Comedy, Children]
 2. 3.923 (4.036) Wallace & Gromit: The Wrong Trousers (1993) [Crime, Animation, Comedy, Children]
 3. 3.917 (3.961) Finding Nemo (2003) [Adventure, Animation, Comedy, Children]
 4. 3.894 (3.921) Toy Story (1995) [Adventure, Fantasy, Animation, Comedy, Children]
 5. 3.884 (4.089) Grand Day Out with Wallace and Gromit, A (1989) [Sci-Fi, Adventure, Animation, Comedy, Children]
 6. 3.838 (3.868) Shrek (2001) [Adventure, Fantasy, Animation, Romance, Comedy, Children]
 7. 3.833 (3.871) Monsters, Inc. (2001) [Adventure, Fantasy, Animation, Comedy, Children]
 8. 3.819 (4.083) Fantastic Mr. Fox (2009) [Adventure, Crime, Animation, Comedy, Children]
```

g) If option 3 is chosen, the user can provide text to search for in the movie titles. For example:



```
Choose from the following menu:
  1 - See top n
  2 - See top n in given genre(s)
  3 - Search movie titles
  4 - quit
 #3
Search in titles for: harry
18 matches. Show #:18
 1. 3.871 (3.989) Harry Potter and the Deathly Hallows: Part 1 (2010) [Action, Adventure, Fantasy, IMAX]
 2. 3.857 (3.914) Harry Potter and the Prisoner of Azkaban (2004) [Adventure, Fantasy, IMAX]
 3. 3.816 (3.910) Harry Potter and the Deathly Hallows: Part 2 (2011) [Action, Adventure, Drama, Fantasy, IMAX, Mystery]
 4. 3.809 (3.888) Harry Potter and the Half-Blood Prince (2009) [Adventure, Fantasy, Romance, IMAX, Mystery]
 5. 3.805 (3.859) When Harry Met Sally... (1989) [Romance, Comedy]
 6. 3.788 (3.862) Harry Potter and the Order of the Phoenix (2007) [Adventure, Drama, Fantasy, IMAX]
 7. 3.762 (3.817) Harry Potter and the Goblet of Fire (2005) [Adventure, Thriller, Fantasy, IMAX]
 8. 3.730 (3.762) Harry Potter and the Sorcerer's Stone (a.k.a. Harry Potter and the Philosopher's Stone) (2001) [Adventure, Fantasy, Children]
 9. 3.658 (3.775) Dirty Harry (1971) [Action, Thriller, Crime]
10. 3.631 (3.875) Deconstructing Harry (1997) [Drama, Comedy]
11. 3.619 (4.500) With a Friend Like Harry... (Harry, un ami qui vous veut du bien) (2000) [Drama, Thriller]
12. 3.586 (3.598) Harry Potter and the Chamber of Secrets (2002) [Adventure, Fantasy]
13. 3.560 (4.000) Harry Brown (2009) [Drama, Thriller, Crime]
14. 3.477 (3.417) Trouble with Harry, The (1955) [Comedy, Mystery]
15. 3.408 (2.000) Let's Get Harry (1986) [Action, Adventure]
16. 3.297 (2.857) Who's Harry Crumb? (1989) [Comedy, Mystery]
17. 3.097 (2.545) Harry and the Hendersons (1987) [Comedy, Children]
18. 2.847 (1.955) Dumb and Dumberer: When Harry Met Lloyd (2003) [Comedy]

Choose from the following menu:
  1 - See top n
  2 - See top n in given genre(s)
  3 - Search movie titles
  4 - quit
 #
```

h) The user can continue choosing from the menu until the user quits.

i) You may assume that the user does not make input mistakes in your program.

Submission

Your project is expected to implement the descriptions above and the following:

- Do not use absolute file locations; use relative locations. Unless otherwise specified, assume input files are in the same working directory as your code files.
- All I/O resources should be properly managed. Catch necessary exceptions and give appropriate / meaningful error messages.
- All class, variable, method, and constant names should follow conventions and be meaningful and accurate.
- Non-obvious parts of your code should be explained high-level with comments.
- Code should be cleanly formatted and indented.
- You must document all of your methods using **proper Javadoc** comments (see 6.14 section in zyBooks to review if needed).

- Create and fill out a README.TXT or README.md file with the following information:
    - AUTHOR(S):
    - STATE OF THIS PROJECT:
        - Describe the state you believe your project is in. (E.g., Is it completely finished and fully functioning and well tested? If not, what state is it in?)
    - KNOWN BUG(S):
        - Describe any issues you are aware that your solution has.
    - PROCESS DESCRIPTION:
        - Describe how the project went for you. (E.g., Did it go smoothly? What obstacles did you face? Which parts did you find easiest/most challenging?)
    - **SPELL CHECKER COLLECTION(S) USED AND WHY:**
        - Describe the collection(s) you chose to use for your spell-checker program and why. If you considered alternatives, describe how you arrived at your decision.
    - **MOVIE RATING COLLECTION(S) USED AND WHY:**
        - Describe the collection(s) you chose to use for your movie rating analytics program and why. If you considered alternatives, describe how you arrived at your decision.

  Save your README file inside your folder containing your project files.

Submission
Zip your entire Project 3 folder (ensuring it contains your README file) and upload it to the Project 3 assignment on Canvas. If you work with a partner, only ***one*** of you should submit your project, ***not*** both.

After you have uploaded the file, double-check to ensure your file was uploaded correctly. It is your responsibility to ensure your submission was done correctly. An improperly submitted project will cost you points.

Additional Exercises (Not required: Not for credit)

- Create a class called **`Composition`**. Read in all of the words in a file called **dictionary.txt**. Each word will be separated by whitespace (i.e. a space, a tab or a newline character). For every word, determine if it can be created by concatenating two other words. For example, the word racecar can be created by concatenating race and car. All such compositions must be output to a file called **compositions.txt** (in the **output** folder) in this format:

  ```
  racecar:race car
  ```

  which is the original word followed by a colon, followed by the first word, followed by a space and finally the second word. Each composition should be on its own line, and they should appear in alphabetical order. For example, `racecar:race car` should be after `another:an other` in the output file.

- Create a class called **`CompositionTwo`**. Read in all of the words in a file called **dictionary.txt**. Each word will be separated by whitespace (i.e. a space, a tab or a newline character). You must read in all words and determine if a word can be created by concatenating three or more words. For example, the word disconsolateness can be created by concatenating disc, on, so and lateness. All such compositions must be output to a file called **moreCompositions.txt** (in the **output** folder) in this format:

  ```
  disconsolateness:disc on so lateness
  ```

  With each answer on its own line. *This is a difficult problem and you may want to use* ***recursion****.*

  Hint: Remember a recursive method can have as many parameters as it needs to keep track of the "current state of the problem." Also, like in the previous problem, the final output is likely to be in alphabetical order.