Administrative

- Team Name: Team JAM
- Team Members: Jeffery James, Maxmilian Pennisi, Anthony Zurita
- GitHub URL https://github.com/mpennisi498/Custom-Spotify-Playlist-Generator
- Link to Video https://youtu.be/tMREhBzNhMA

Extended and Refined Proposal

Problem

We aim to address the challenge of having access to an enjoyable and modern playlist of songs by curating quality, custom-made playlists that prioritize the inclusion of the most popular and trending songs for everyday individuals. Playlists are mainly auto suggested by music providers that predict what type of music you would like to listen to. However, these providers don't have an easy playlist builder option where users can just input certain parameters and build their own custom playlist.

Motivation

The motivation for solving this problem is that in people's busy lives, they don't have the time to make a personalized playlist based on the preferences they want. They willingly accept pre-generated playlists or have to search up individual songs, which can be a hassle and most of the time ends in not having an enjoyable music experience.

Features

Our program will allow the user to filter between genres, artists, explicit or clean, and number of songs to generate a custom playlist for them. All filters are optional besides the number of songs. All songs have a popularity rank, which determines whether they are picked over a different song. Our data structures and algorithms sort the songs based on the filters applied and additionally picks the most popular songs that fit that criteria, allowing users to have custom playlists with the most trending songs. This will allow people to create custom and popular playlists for different events and parts of their life, such as for daily workouts, special parties, and other relevant events.

Tools/Languages/APIs/Libraries Used

The languages we used were Python and Javascript. We used Python to create the data structures and handle processing all of the data. Javascript was used to create the frontend and to display results. The libraries within python we used were Pandas to read and store the Spotify Data, flask to host the backend server, and Spotipy to handle the API request to Spotify. In the frontend we used React and Typescript as our preferred framework. For the UI elements we had React Bootstrap to have easy customizable components, and React Spotify Embed to easily embed the playlist into the frontend.

Algorithms Implemented

The driving algorithm behind our playlist generator is a filtering algorithm. First, the algorithm starts by passing in the data set with 100,000+ data points to the MaxHeap and Red/Black Tree data structures. After, there are two paths one for the Max Heap and one for the Red/Black Tree. For the Max Heap path, an ExtractMax() function is called and the node retrieved is

compared to the user-applied filters. If the node passes all the user applied filters, the node's song ID value is appended to an array, so it can be added to the final playlist later. For the Red/Black Tree path, the data points are balanced using red black tree properties.Then the search function is called where songs that match the user specified parameters are passed into a set, which is returned to the frontend. In the end whether the MaxHeap or Red/Black Tree is used, the array with song IDs is pushed to the Spotify API to create the custom playlist.

Additional Data Structures/Algorithms Used

Apart from the Max Heap and the Red/Black Tree, we also used the set, hash table, and array data structures.

In the implementation of MaxHeap the array and hash table data structures were used. The array data structure was used to implement the MaxHeap as an array. The hash table (also known as a Dictionary in Python) was used to represent each node. For example, at any valid index in the array for the Max Heap there would be a dictionary with the keys being 'popularity', 'ID', 'artistName', 'genre', and 'explicit' along with their associated values (taken from data set). The Dictionary data structure helped store all the needed information about a song at all indexes in the array.

A set was used in the Red Black Tree implementation to ensure that duplicate songs did not appear in the playlist. The main search function is supported by three helper functions that search for each parameter the user can select. The results from these helper functions may overlap. Therefore a set was needed to maintain the uniqueness of the playlist.

Distribution of Responsibilities and Roles

Team Member	Responsibility/Role
Max Pennisi	● Implement front end and integration with backend data structures
Jeffrey James	● Create Red/Black Tree data structure in Python
Anthony Zurita	● Create Max Heap data structure in Python ● Implement common MaxHeap operations such as Insert(), ExtractMax(), along with helper functions

Analysis

Changes

One slight change was made after we submitted the proposal. Since the proposal was general and we did not know the specifics and intricate details of our filtering algorithm, we later decided to use the popularity rank as a driving factor to help pick and choose songs for all custom generated playlists. This makes it so that we prioritize adding in the most popular songs on top of the user-applied filters. This makes it so that users will have custom playlists that also have the most popular and trending songs too.

Big O Worst Case Time Complexity Analysis

Main Function Time Complexity
The addData() function loops through the end of the data set which is an O(n) operation where n is the number of songs inside of the data. Each song is then inserted into a heap and Red Black Tree which is an O(logn) operation. Since the loop and insertion are dependent on each other the time complexity for inserting into both data structures is O(nlogn + nlogn) = O(nlogn).

The time complexity of the makePlaylist() function is O(n). Where n is the number of songs in the dataset. The function takes in an array of tracks and uses the Spotify API to create a playlist. In the worst case scenario all tracks in the dataset are passed into the function and to a playlist and no filters are applied.

The time complexity of the main function returnPlaylist() is O(2(nlogn) + n+ n^2). Where n is the number of songs in the dataset. The main function calls the addData() function which is O(nlogn). After that it searches the rb tree and filters the heap. Extracting the max of the heap is O(1) and with the Heapify Down function needed to be called its becomes O(logn) but in the worst case scenario each node of the heap is popped off n times making it O(nlogn). All filters are O(1) operations. For the red black tree, the filtering is done within the data structure and is an O(n) operation. In the worst case that function is called n times so filtering and searching the red black tree becomes an O(n^2). Finally it calls the makePlaylist() function which is O(n). Coming together as O(2(nlogn) +n + n^2) simplifies to O(n^2). Where n is the number of songs in the dataset in the worst case scenario.

Max Heap Time Complexity
The time complexity of insertNode() is O(logn), where n is the number of nodes in the MaxHeap. The insert function creates a dictionary which is O(len(d)), where d is the dictionary. Since the dictionary created is always a length of 5 it can be simplified to O(1). Appending a node to the end of the list is O(1). Lastly, a helper function, HeapifyUp(), is called, which is O(logn), where n is the number of nodes in the MaxHeap. This is because in the worst case the node will have to HeapifyUp() the height of the tree, which has a logarithmic relationship with the number of nodes, resulting in a final time complexity of O(logn).

The time complexity of extractMax() is O(logn), where n is the number of nodes in the MaxHeap. The extractMax function assigns the variable popped (which is returned at the end of the function) to the root item by doing popped = self.heap[0], which is O(1). After reassigning the last item in the MaxHeap to be the root (an O(1) operation), then the last item in the array is deleted, which has a time complexity of O(1). This is because the item being deleted is always the last one in the array, rather than being a random index in the array. Lastly, a helper function, HeapifyDown(), is called, which is O(logn), where n is the number of nodes in the MaxHeap. This is because in the worst case the node will have to HeapifyDown() the height of the tree, which has a logarithmic relationship with the number of nodes, resulting in a final time complexity of O(logn).

Red Black Tree Time Complexity

Search: The search function has a time complexity of O(n) where n is the number of nodes in the tree. The search function calls the helper functions searchGenre,searchArtist and searchExplicit. Each of these functions go through the whole tree to find songs that match their parameters making each of them O(n). These helper functions also add each song to an array which has an amortized time complexity of O(1) . In the search function, there is a for loop that goes through each node in the array and checks which parameters it satisfies, and if the condition is met, the node is added to a set. Sets are implemented using hash tables in python, meaning the time complexity of adding to a set is O(1). So the time complexity of the for loop and the code within the for loop is O(n).

RB Tree Balancing: The worst case time complexity of the rbTreeBalance function is O(log(n),where n is the number of nodes in the tree. The balancing function does two things: restructures the nodes and changes their colors to satisfy red black tree properties. Restructuring rotateLeft and rotateRight have an O(1) time complexity because they both involve reassigning a finite number of pointers each time. The recoloring is O(log(n),where n is the number of nodes in the tree. Changing the colors itself is an O(1) operation because it involves changing the value of the variable, but their can be situations where it is needed to recolor farther up the tree, and the worst case would involve fixing a double red situation along the entire path from the newly inserted node to the root node making the overall time complexity of recoloring Olog(n).

Insert: The worst time complexity of the insertNode in RB Tree is Olog(n) , where n is the number of nodes the RB Tree has. The reason for the logarithmic time is that the tree always remains balanced after every insertion, so each comparison between the passed in node and the nodes existing in the tree approximately halves the number of nodes needing to be checked exhibiting a logarithmic relationship.

Reflection

Group Experience

As a group, we all had fun developing this project. We all had to review our python skills which was useful, along with the way the frontend and backend functionality came out. We are very happy with the way our project came out and are excited to show it off.

Challenges

The biggest challenge for the backend data structures was the team having to refresh their memory on Python and its syntax. With the frontend, the largest challenge was figuring out the API with Spotify and figuring out the Spotipy package. It took multiple hours to figure out how to send song requests through Spotify's API to create a custom playlist.

If we were to restart

If we had the opportunity to restart the group, we could've added more features to tailor the playlist even more to what a user would want in their playlist. Additionally, we think we should have spent more time looking for a larger dataset as it would be more useful and relevant to

have more song options to choose from even though the playlist would take longer to generate, it would have led to a higher quality playlist.

What We Learned

Max: I was already very familiar with React and Typescript and bootstrap coming into this project, but I never learned Flask. So I spent time learning to set up Flask and be able to connect it to the Frontend through http requests. I also had never used the Spotipy library or the Spotify developer tools and API. So I learned how to use those libraries by reading the docs and watching videos online. Additionally, when working with a group I learned the importance of keeping all team members on the same page

Anthony: I used to be more familiar with Python back in programming 1, but when doing this project I needed to look back at past assignments and other resources to refresh my memory on how classes work in Python. After some time I was able to get the hang of it. I learned that once you learn programming concepts such as classes, no matter the language you will still be able to program. Implementing a MaxHeap from scratch helped me gain a better understanding of its key features and properties. I also learned how to use the Pandas library in Python to parse the large data set we used for the custom playlist generator.

Jeffrey: It was useful to go back to some of the Python concepts learned in Prog 1 and implement a topic learned in Data structures.I learned how to keep my code modular for ease of use when interacting with a UI and other libraries. I also gained a deeper understanding of Red Black tree properties and balancing in Red Black trees. I also learned how data is passed to an API from backend functions.

References

Frontend + Flask:
Creating a Flask React App: https://www.youtube.com/watch?v=7LNl2JlZKHA&t=74s
React Bootstrap Docs: https://react-bootstrap.netlify.app/docs/components/accordion
Pandas Docs: https://pandas.pydata.org/docs/
Spotipy API Tutorial: https://www.youtube.com/watch?v=jSOrEmKUd_c
MaxHeap Implementation Resources:

- https://www.geeksforgeeks.org/max-heap-in-python/
- https://www.geeksforgeeks.org/python-dictionary/
- https://www.geeksforgeeks.org/floor-ceil-function-python/
- https://www.geeksforgeeks.org/complexity-cheat-sheet-for-python-operations/
- https://www.geeksforgeeks.org/python-remove-rear-element-from-list/
- Discussion 7 - Exam 1 Review slides

Red Black Tree Resources:

- https://blog.boot.dev/python/red-black-tree-python
- https://www.youtube.com/playlist?list=PL9xmBV_5YoZNqDI8qfOZgzbqahCUmUEin
- https://pages.cs.wisc.edu/~skrentny/cs367-common/readings/Red-Black-Trees/
- https://stackoverflow.com/questions/7351459/time-complexity-of-python-set-operations