Media File Retrieval: How to Reduce Time Complexity

By

James Thomas

Design and Analysis of Algorithms: CS627

Colorado Technical University

08/06/2023

Table of Contents

[Introduction 3](#_Toc142252179)

[Divide-and-Conquer Algorithm 3](#_Toc142252180)

[Time Complexity 5](#_Toc142252181)

[References 7](#_Toc142252182)

# Introduction

Our primary objective is to enhance the productivity of mobile devices amidst the ever-evolving technological environment. As a part of this ongoing endeavor, our dedicated team has undertaken a very important task. We must reduce the time it takes to retrieve media from playlists (Jiao, Y. 2021). Leading this new project our Algorithm Group will be breaking down key details to emphasize the importance of divide-and-conquer algorithms and how they will resolve the problem of increased time, in retrieving media from music playlists. With the target of changing how mobile devices interact with our music application, our team has embarked on a two-part assignment that involves the implementation of a sophisticated search algorithm in Python code. This algorithm is designed to locate specific songs within a playlist. This algorithm will comprise an array of strings that are meticulously sorted in alphabetical order. The algorithm takes two essential inputs: the playlist and the target song. If successful in its search, the algorithm will return the index of the song; otherwise, it will signify its findings by returning -1.

We will continue to discuss further the intricacies of Big-O notation and compare it to the established binary search approach in the additional sections after reviewing the developed code(Ahmad & Jawawi 2011). By exploring and reviewing these aspects, we hope to bring forward ideas on the potential advancements that our divide-and-conquer search algorithm might bring to the field of media and playlist optimization. We explore this in the hope to contribute to ongoing endeavors for excellence in mobile device performance and user experience.

# Divide-and-Conquer Algorithm

To review the algorithm below, we will look line by line, and break down the divide and conquer code, analyzing its step-by-step implementation and the utilization of the ternary search approach. In the first line, we define the function name “find” that takes a list of songs and a target title as input. In lines 2 and 3, we initialize pointers “being” and “end” to the start and end indices of the list. On line 5, we initialize that -1 indicates that the title was not found. On line 7, we continue to start a “while loop”. In this while loop, the algorithm calculates the first mid-point “mid 1” and the second mid-point “mid2” through the trisection search. A trisection search referring to the act of dividing something into three equal or nearly equal parts. The difference here from the term “Ternary” is that Ternary refers to a system or concept that involves three elements or options.

As the loop continues , it checks to see if the target title is less than or greater than different mid sections of the list portion. The code here does not look directly at the elements in the array or in this case, songs in a playlist, but it is considering the spaces in between each element as a means to section multiple variables. At the end of the loop, we add “return index” to return the index where the title was found, or return -1 if it was not found.

.01)def find(songs, title):

.02) begin = 0 #Initialize pointers 'begin' and 'end' to the start and end indices of the list

.03) end = len(songs) - 1

.04) # Initialize 'index' to -1, indicating that the title is not found initially

.05) index = -1

.06) # Start a loop to perform the trisection search within the range of indices

.07) while begin <= end:

.08) # Calculate the first mid-point 'mid1' for trisection

.09) mid1 = ((end - begin) // 3) + begin

.10) mid2 = (2 \* ((end - begin) // 3)) + begin

.11) if title < songs[mid1]:

.12) end = mid1 - 1

.13) # Check if the target title is greater than the song at 'mid1'

.14) elif title > songs[mid1] and title < songs[mid2]: # and less than the song at 'mid2'

.15) end = mid2 - 1

.16) begin = mid1 + 1

.17) # Check if the target title is greater than the song at 'mid2'

.18) elif title > songs[mid2]:

.19) begin = mid2 + 1

.20)

.21) elif title == songs[mid1]:

.22) index = mid1

.23) break

.24)

.25) elif title == songs[mid2]:

.26) index = mid2

.27) break

.28) # Return the index where the title was found (or -1 if not found)

.29) return index

.30)

.31) # List of song titles for testing

.32)my\_songs = [

.33) 'Chop Suey',

.34) 'Drowning',

.35) 'Eleanor Rigby',

.36) 'Fye Fye',

.37) 'Ghost of You',

.38) 'Holy Mountain',

.39) 'IEAIAIO',

.40) 'Jump',

.41) 'Killer Queen',

.42) 'Look @ This',

.43) 'My Girl',

.44) 'Never too late',

.45) 'One',

.46) 'Peppers and Onions'

.47)]

.48)# Call the 'find' function with the list of songs and target title “One”

.49)# Print the result, which should be the index of the target title in the list

.50)print(find(my\_songs, 'One'))

# Time Complexity

Now that we have reviewed how the algorithm works, we will now review the time complexity of the ternary search algorithm. The algorithm divides the range of elements into three parts using “mid1” and “mid2”. The algorithm itself then performs comparisons and adjusts the pointers “begin” and “end” based on the comparison results. The number of iterations required to narrow down the search range is proportionate to the logarithm base 3 of the size of the input list. The algorithm divides the range of elements into three parts in each iteration of the “while loop” and the number of iterations required to reach a single element range (Ahmad & Jawawi 2011). Therefore the time complexity of this algorithm can be expressed as O(Log3N), where N is the size of the input list that is our playlist consisting of songs.

Our next question is, how does a Ternary search algorithm like this compare to a Binary search algorithm? A Ternary search has an expression that appears as O(Log3N), whereas a Binary search has a time complexity expression reflected as O(Log2N) (Ahmad & Jawawi 2011). A binary search is more efficient than a ternary search in terms of time complexity. This is because the binary search range is in half in each iteration, resulting in a faster convergence to a single element range or determining that the element is not present (Bajwa, et al 2015). The logarithm base 2 in binary search is more efficient in terms of time complexity compared to the logarithm base 3 in ternary search. Ternary search algorithms hold advantages in convex functions, narrowing search space and reducing the number of iterations, which can also, reduce time to retrieve media files, while also retaining the additional benefits (Bajwa, et al 2015).

To conclude, we have designed and implemented a search algorithm in Python that efficiently locates a specific song within a sorted playlist of strings. The algorithm takes advantage of a divide-and-conquer strategy, reminiscent of the binary search approach. However, we have introduced a variation that divides the list into three sublists at each iteration, enhancing the efficiency of our search. Our trisection-based search algorithm efficiently locates songs within a sorted playlist, creating balance in the simplification of the algorithm and time complexity. This approach is innovative, dividing the list into three sub-lists, and showcasing how minor variations to established algorithms can yield significant performance improvements. This makes the algorithm a valuable addition to the repertoire of search algorithms.

# References

Ahmad, N. B. H., & Jawawi, D. N. A. (2011). Algorithm Efficiency Analysis.

Bajwa, M. S., Agarwal, A. P., & Manchanda, S. (2015, March). Ternary search algorithm: Improvement of binary search. In 2015 2nd International Conference on Computing for Sustainable Global Development (Indiacom) (pp. 1723-1725). IEEE.

Jiao, Y. (2021). Digital music waveform analysis and retrieval based on feature extraction algorithm. Advances in Multimedia, 2021, 1-10.

Levitin, A. (2016). Introduction to the Design and Analysis of Algorithms (3rd ed.). Pearson Learning Solutions. <https://bookshelf.vitalsource.com/books/9781323417638>