Dynamic Programming: Image Comparison App

By

James Thomas

Design and Analysis of Algorithms: CS627

Colorado Technical University

08/20/2023

Table of Contents

[Introduction 3](#_Toc143533466)

[Pseudocode Design for Image Comparison 3](#_Toc143533467)

[Optimality of Dynamic Programming 7](#_Toc143533468)

[Time Complexity of the Algorithm 8](#_Toc143533469)

[Portfolio of Previous Projects 8](#_Toc143533470)

[In-Place Audio Reversal Algorithm: Analysis and Implementation 9](#_Toc143533471)

[Base Code for Sort 9](#_Toc143533472)

[Time and Space complexity 10](#_Toc143533473)

[True Runtime Function 11](#_Toc143533474)

[Media File Retrieval: How to Reduce Time Complexity 13](#_Toc143533475)

[Divide-and-Conquer Algorithm 14](#_Toc143533476)

[Time Complexity 16](#_Toc143533477)

[Greedy Algorithm: Optimizing Disk Allocation 17](#_Toc143533478)

[Greedy Algorithm Pseudocode 18](#_Toc143533479)

[Optimality of Algorithm and Time Complexity 20](#_Toc143533480)

[Options of Brute Force Algorithm and Time Complexity 21](#_Toc143533481)

[References 22](#_Toc143533482)

# Introduction

Our algorithms group has been presented with an important task in the development of an innovative image-processing application. The primary objective of the application is to perform with sophistication and compute operations on monochromatic images. The application’s core functionality centers around the comparison of multiple black-and-white images. Even though there are multiple paths to perform the objectives of the application, our team has reached a consensus that utilizing dynamic programming will be the best way to complete the task at hand by providing fast and refined solutions.

Our team proposes to design and implement an app that efficiently analyzes and contrasts black-and-white images using dynamic programming principles. By utilizing dynamic programming our application will deliver exceptional performance, catering to both speed and precision. Our comprehensive approach will involve algorithmic intricacies and code optimization. The ultimate goal is to craft a versatile tool that meets the exacting demands of image analysis, catering to a range of potential applications from quality control to creative design.

# Pseudocode Design for Image Comparison

The provided code defines the functions for calculating edit distances between strings and comparing images based on these distances. The code begins by declaring a dictionary that we named “edit\_distance\_already\_done”, which stores the precomputed edit distances of pairs of strings. This dictionary will be utilized to optimize the calculation of edit distances in subsequent operations. Thereafter, a function named “edit\_distance” is defined and responsible for computing the edit distance between two input strings, “s1” and “s2”. The function employs a recursive approach with dynamic programming, checking if the edit distance for the given pair of strings has already been computed and stored in the “edit\_distance\_already\_done” dictionary. If the edit distance is already available, the function directly returns the stored value. Otherwise, the function initializes a list named “return\_value” to hold the calculated results. It then proceeds to handle carious base cases: if the input strings are identical or if one of them is empty. Then, depending on the conditions set, the function constructs the “return\_value” list by recursively calculating the edit distances for cases. An example of such cases would be character changing, insertions in one of the strings, and then determining the operation with the smallest edit distance. In the end, the calculated “return\_value” is stored in the “edit\_distance\_already\_done” dictionary for future reference, and the function returns the computed result.

Pseudo Code:

1.) Initialize an empty dictionary called `edit\_distance\_already\_done`.

2.) Create a function named `edit\_distance` that takes two strings, `s1` and `s2`, as inputs.

3.) Check if `(s1, s2)` is a key in `edit\_distance\_already\_done`

4.) if so, return its corresponding value.

5.) Initialize an empty list `return\_value`.

6.) if `s1` is equal to `s2`

7.) set `return\_value` as an empty list.

8.) Else if `s1` is an empty string

9.) set `return\_value` as a list containing "Additional letter" messages for each character in `s2`.

10.) Else if `s2` is an empty string

11.) set `return\_value` as a list containing "Additional letter" messages for each character in `s1`.

12.) Otherwise

13.) split `s1` and `s2` into head and tail, and proceed with comparisons.

14.) if the heads are equal

15.) recursively calculate the edit distance for the tails.

16.) if the heads are not equal

17.) generate different options for changes and insertions

18.) compare their lengths, and set `return\_value` accordingly.

19.) Store `return\_value` in `edit\_distance\_already\_done` and return it.

20.) Define a function named `compare`

21.) that takes three inputs: `initial\_image`, `final\_image`, and `threshold`.

22.) Initialize a variable `total\_amount\_of\_changes` to track the cumulative changes.

23.) Iterate through the rows of `initial\_image` and `final\_image`.

24.) Calculate the edit distance between the rows and add it to `total\_amount\_of\_changes`.

25.) if `total\_amount\_of\_changes` is less than the `threshold`

26.) return "The images are similar";

27.) otherwise

28.) return "The images are different".

29.) Define `initial\_image` and `final\_image` as lists of strings representing image rows.

30.) Call the `compare` function with `initial\_image`, `final\_image`, and a specified `threshold`.

31.) Print the result of the comparison.

After the “Edit” portion of the code, the code also defines a function “compare”, that takes three arguments: “initial\_image”,”final\_image”, and “threshold”. This function aims to compare two images represented as lists of strings by calculating the edit distances between corresponding rows and determining whether the total amount of changes falls below the specified “threshold”. Within the function, a counter named “total\_amount\_of\_changes” is initialized to keep track of cumulative changes across all rows. The function iterates through each row of the input images, calculates the edit distance between the corresponding rows using the ”edit\_distance” function, and increments the “total\_amount\_of\_changes” counter by the computed amount of changes. After processing all rows, the total amount of changes is compared to the provided threshold. If the total changes are similar; otherwise, it returns a string indicating that the images are different.

After the “Compare” section of the code, two sets of initial and final images are defined as lists of binary strings entered to test and demonstrate the code’s functionality. The “compare” function is then called with the provided initial and final images, as well as a threshold value of 20. The result of the comparison is printed to the console, indicating whether the images are similar or different based on the calculated edit distances and the specified threshold.

Python code :

1.) # A dictionary to store the calculated edit distances for pairs of strings

2.) edit\_distance\_already\_done = {}

3.)

4.) # Function to calculate the edit distance between two strings

5.) def edit\_distance(s1, s2):

6.) # Check if the edit distance for this pair of strings has already been calculated

7.) if (s1, s2) in edit\_distance\_already\_done:

8.) return edit\_distance\_already\_done[(s1, s2)]

9.)

10.) # Initialize the return value list

11.) return\_value = []

12.)

13.) # Base cases: if the strings are the same or one of them is empty

14.) if s1 == s2:

15.) return\_value = []

16.) elif len(s1) == 0:

17.) return\_value = [f'Additional letter {x}' for x in s2]

18.) elif len(s2) == 0:

19.) return\_value = [f'Additional letter {x}' for x in s1]

20.) else:

21.) # Break down the strings into their first characters and remaining parts

22.) head\_s1 = s1[0]

23.) tail\_s1 = s1[1:]

24.) head\_s2 = s2[0]

25.) tail\_s2 = s2[1:]

26.)

27.) # If the first characters are the same, move to the next characters

28.) if head\_s1 == head\_s2:

29.) return\_value = edit\_distance(tail\_s1, tail\_s2)

30.) else:

31.) # Calculate the edit distances for different operations (change, insert)

32.) change = [f'{head\_s1} was changed to {head\_s2}'] + edit\_distance(tail\_s1, tail\_s2)

33.) s1\_insert = [f's1 has additional {head\_s1}'] + edit\_distance(tail\_s1, s2)

34.) s2\_insert = [f's2 has additional {head\_s2}'] + edit\_distance(s1, tail\_s2)

35.)

36.) # Determine the operation with the smallest edit distance

37.) if len(change) <= len(s1\_insert) and len(change) <= len(s2\_insert):

38.) return\_value = change

39.) elif len(s1\_insert) <= len(change) and len(s1\_insert) <= len(s2\_insert):

40.) return\_value = s1\_insert

41.) elif len(s2\_insert) <= len(change) and len(s2\_insert) <= len(s1\_insert):

42.) return\_value = s2\_insert

43.)

44.) # Store the calculated edit distance and return the value

45.) edit\_distance\_already\_done[(s1, s2)] = return\_value

46.) return return\_value

47.) # Function to compare two images using their edit distances

48.) def compare(initial\_image, final\_image, threshold):

49.) # Initialize a counter for the total amount of changes

50.) total\_amount\_of\_changes = 0

51.)

52.) # Iterate through each row in the images

53.) for i in range(len(initial\_image)):

54.) initial\_row = initial\_image[i]

55.) final\_row = final\_image[i]

56.)

57.) # Calculate the edit distance between the rows and count the changes

58.) amount\_of\_changes = len(edit\_distance(initial\_row, final\_row))

59.) total\_amount\_of\_changes += amount\_of\_changes

60.)

61.) # Compare the total changes to the threshold and return the result

62.) if total\_amount\_of\_changes < threshold:

63.) return 'The images are similar'

64.) else:

65.) return 'The images are different'

66.)

67.) # Initial and final images to compare

68.) initial\_image = [

69.) '010010',

70.) '100001',

71.) '011110'

72.)]

73.)

74.) final\_image = [

75.) '010011',

76.) '100001',

77.) '011110'

78.)]

79.)

80.) # Call the compare function and print the result

81.) print(compare(initial\_image, final\_image, threshold=20))

# Optimality of Dynamic Programming

The dynamic programming solution derived in the code is a variation of the “Edit Distance” problem, which seeks to determine the minimum number of operations required to transform one string into another. While the given code doesn’t aim for minimality, it rather tried to find a sequence of operations to convert one string into another with a specific focus on “edit” operations. Including changing, inserting, or deleting characters through out. The solution’s optimality can be discussed in terms of its structure. The algorithm exhaustively explores all possible combinations of edit operations to find the sequence that produces the least number of changes. Through utilization of the dictionary function, we store computed results in the “edit\_distance\_already\_done” portion of the code. By using this form of memorization the algorithm avoids recalculating the edit distance for the same pair of substrings, significantly reducing redundant calculations. This memorization step ensures that the solution achieves optimal substructure, a key property of dynamic programming.

# Time Complexity of the Algorithm

In terms of the time complexity, X will be represented the length of the first string and Y will represent the length of the second string. The dynamic programming solution will need to be traversed through all possible combinations of substrings of both strings, resulting in the time complexity of O(IJK). In this case, I, J, and K represent dimensions or sizes related to the problem instance. Given the two strings of equal length “n” to determine their edit distance, a maximum of O(n2) operations is required. However, this scenario changes when the strings have distinct lengths, denoted as “n” and “m”. In such cases, the computational complexity for calculating the edit distance escalates to O(nm). The principle extends to situations where the compared entities possess dissimilar row lengths, in comparison to the images. Specifically, if one row consists of ‘J’ elements and another row has ‘K’ elements, the computation of the edit distance for a single row necessitates O(JK) operations. When applying this process across “I” rows, each requiring edit distance calculations, the cumulative time complexity becomes O(IJK).

# Portfolio of Previous Projects

In this portfolio section, we will showcase a collection of our previous projects that reflect the diversity and depth of our algorithmic expertise. Each project highlights our commitment to crafting innovative solutions that address unique challenges. These projects stand as testaments to our team's dedication to innovation and problem-solving across various domains. The repository includes an analysis and steps for implementing an in-place audio reversal algorithm, media file retrieval and reducing algorithm and a review of greedy algorithms and disk allocation.

# In-Place Audio Reversal Algorithm: Analysis and Implementation

We have often noticed that sorting algorithms have frequent applications in business and technology. Optimizing memory and space usage is crucial for applications to achieve better efficiency with a lower cost. This instance is especially true for applications that process audio files, as these files can be large and require a significant amount of memory to store. This document will review and analyze elements related to time and space efficiency using Python code for sorting audio content in reverse order.

# Base Code for Sort

The code below contains an algorithm designed to reverse the array after the function is entered and the request is initiated.

1. def reverse (array):

2. size=len(array)

3. for i in range(size//2): #defines the i in the bottom half of the array

4. j = size-1-i #finding the corresponding value to swap

5. array[i], array[j] = array[j],array[i] #swaps values

In the first line there is a function that “defines” the process as “reverse” and that we are creating this code with an “array” as mentioned in parentheses. In line 2, we define the size of the array. In line 3, we define the “ i ” in the bottom half of the array. In line 4, “ j ” is the identified for finding the corresponding value to swap. Line 5 is where we execute the swap of elements in the array (Langtangen, 2008).

A diagram of a diagram

Description automatically generatedIn order to visualize line 5 as a function, Figure 1. is a diagram that shows an example with 6 integers. The first three integers would be represented by the “ i ” variable while the last three integers are represented by the letter “ j ” (Langtangen, 2008).

Figure 1.

# Time and Space complexity

To analyze the time complexity of the code, we must review the worst-case time complexity. It is more common to define an algorithm’s worst-case time complexity to guarantee the upper bound of the running time, which in turn, helps in designing reliable and efficient algorithms (Levitin, A. 2016). In this code, the first half of the array swaps with the corresponding elements. The number of iterations depends on the size of the array, which is “N.” The “for loop” iterates “size//2” time, or in the form of an equation: Number of iterations = (N/2). Inside the loop, the code reflects constant time operations and, to be more specific, “ j,” for example. The algorithm continues to swap these particular elements. With these instances in mind, understanding the “for loop” and the constant time operations, we can conclude the time complexity of the “reverse” algorithm as O(N/2). With this in mind, it is more common to consider the dominant factor, which reflects approximated values of how the time complexity changes with large input sizes. The constant factor of “/2” or “1/2” is not significant compared to the linear factor of size, which is “N.” When using big O notation, the constant factors are not taken into account. Therefore, the final time complexity of the algorithm is O(N).

In terms of the space complexity of the algorithm, we will review the memory locations. The input array occupies the “size” memory locations where “Size = len(array)” or, in other words, where size is the length of the array. In the instance of the “for loop,” there are two additional elements: “ i ” and “ j,” which represent one memory location each. The information provided shows that our total number of memory locations can be “Size + 1 + 1 = Size + 2”. Because the variables in the code are constants in amounts of memory, the space complexity of this algorithm is O(1). If we take “Size + 2” and remove the constant, we would be left with O(size) or, in other words: O(1).

# True Runtime Function

To accurately measure the algorithm's runtime, we have a function that efficiently helps to determine the true time :

# Import the `timeit` module to measure the execution time of the `reverse` #function.

import timeit

# Create an input array `x1` containing integers from 0 to 99 using the `range` #function and convert it to a list.

x1 = list(range(100))

# Measure the execution time of the `reverse` function with input array `x1`.

# The `timeit.timeit` function runs the `reverse` function once (number=1) and #returns the time taken in seconds.

# It uses a lambda function to call `reverse(x1)` as the code to be timed.

print(timeit.timeit(lambda: reverse(x1), number=1))

x2 = list(range(1\_000))

print(timeit.timeit(lambda: reverse(x2), number=1))

x3 = list(range(10\_000))

print(timeit.timeit(lambda: reverse(x3), number=1))

x4 = list(range(100\_000))

Figure 2

|  |  |
| --- | --- |
| N | Time |
| 100 | 2.33E-05 |
| 1000 | 0.000122 |
| 10,000 | 0.003317 |
| 100,000 | 0.023165 |
| 1,000,000 | 0.228669 |
| 10,000,000 | 2.741688 |

print(timeit.timeit(lambda: reverse(x4), number=1))

x5 = list(range(1\_000\_000))

print(timeit.timeit(lambda: reverse(x5), number=1))

x6 = list(range(10\_000\_000))

print(timeit.timeit(lambda: reverse(x6), number=1))

In this code, multiple and varying sizes of arrays were used. The varying sizes are represented in Figure 2 as “ N ” and the column of times adjacent where the output of seconds is provided after calling the function (Agha & Nawaz 2021). A Cartesian plane was created on an Excel spreadsheet, and a plot was populated after entering these details into the plot seen in Figure 3.

# Media File Retrieval: How to Reduce Time Complexity

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.

# Greedy Algorithm: Optimizing Disk Allocation

In the world of multimedia and mobile technology, the challenge of transferring substantial media files from older storage disks to newer disks has become a critical and difficult task for our company. While the initial approach of transferring files randomly is working for now, I am convinced that there exists a more efficient and strategic method to optimize storage space utilization on the new disks. The prospect of achieving such efficiency has motivated us to explore the design of a greedy algorithm that not only streamlines the media transfer process but also minimizes wasted storage capacity across the array of new disks (Schuetz & Caflisch 2008). This endeavor is rooted in the optimization of problems, where the goal revolves around maximizing or minimizing a specific quantity within a set of constraints. In our case, the optimization challenge centers around effectively organizing a collection of “n” media files – each with varying sizes – onto a set of “m” disks, each possessing its storage capacity.

The primary objective here is to minimize the unused storage space on each disk, ensuring efficient resource utilization. This will reduce the overall number of disks required for the storage process. This would also contribute to cost savings across the board. While striving for optimal distribution, the focus will be placed on filling each disk as comprehensively as possible, leaving minimal unutilized space. The pinnacle of success would involve perfectly aligning each disk’s usage, leaving no wasted storage space. In the events where some disks remain unused, we stand to gain, as these surplus disks can be returned for a refund, adding a financial incentive to our optimization goals and milestones. Through the synergistic behaviors of intelligent algorithms, innovative thinking, and the drive to redefine efficiency, our organization is on the brink of transforming the way we manage the storage of multimedia content (Kuhnle, A. (2019).

# Greedy Algorithm Pseudocode

The provided code is written in a programming language that appears to be a pseudocode or a general form of language that can apply to many (Horowitz, et al.1997). The code describes a greedy algorithm for storing 'n' files onto 'm' disks based on file sizes 's' and disk storage capacities 't'. The goal is to place the largest files on the disks in a way that optimizes space usage.

Starting from the function “DiskTransfer”, we take the four parameters: “n” (number of files), “m”(number of disks), “s” (array of file sizes), and “t” (array of disks storage capacities). In sorting files by size, the algorithm starts by sorting the file sizes in a “non-decreasing order”. This is done using the “Heap Sort” algorithm(which is not completely shown in the pseudocode). The sorted file sizes are sorted in the “sortS” array (Sharma, et al. 2008). The Algorithm then creates the “Reverse Map” by associating each sorted file size with its original file size. This map will be used later to determine the original size of a file from its sorted size. After the mapping process, the code then enters the “greedy approach” through the assignment of files onto disks (Schuetz & Caflisch 2008). It iterates each file and attempts to assign the file to the lowest indexed disks that have sufficient storage space. After all, files have been assigned to disks, the algorithm returns the mapping that indicates which file is sorted on each disk. Here's the code with comments simplified for better understanding:

1.) //Stores n files onto "m" disks with file size "s" and disk storage "d"

2.) // This solution simply chooses the largest file to be sorted at each iteration

3.) //stores it on //the lowest indexed disk possible

4.)

5.) DiskTransfer (n,m,s,t)

6.)

7.) //Since we are using greedy approach, it'll make sense to sort out files by size

8.) //here we assume the SortingObject contains a sorted list

9.) SortingObject.sortedList and an

10.)

11.) SortingObject = HeapSort(s);

12.) // Get the sorted list of file sizes

13.) sortedS = SortingObject.sortedList;

14.)

15.) // Create map from sorted file sizes to Original sizes

16.) map\_from\_sortedS\_to s = sorted.reverseMap;

17.)

18.) //In a greedy fashion, assign each file to the lowest index possible

19.) for i := 1 to n do { //iterate through the files

20.) for j := 1 to m{ //iterate through the disks

21.) if sorted[i]<t[j]

22.)

23.) // Add the file to the disk and adjust the maximum storage

24.) map[map\_from\_sortedS\_s[i]] = j //add to the disk

25.) t[j]-sorted[i]; //adjust new maximum storage

26.)

27.) //Since we have sorted the file onto a disk

28.) //successfully, we can break from the inner "for" loop

29.) break;

30.) } end if

31.) } end for

32.) } end for

33.) Return map;

The code describes a greedy algorithm that sorts the files based on their sizes and then assigns each file to the disk with the lowest index that has enough storage capacity. It iterates through the files and disks, to check if the current file can fit into the surplus storage of a disk. If the files can fit the space, the file is assigned to that disk, and the storage capacity of the disk is adjusted. The process continues until all files are assigned. The goal of this algorithm is to efficiently distribute files among disks to minimize wasted space and make the best use of available storage. The code uses “Heap Sort” and greedy assignment to achieve this optimization. The algorithm sorts the files by size, then iterates through each file and assigns it to the lowest-indexed disk that can accommodate it. The greedy approach may not always produce an optimal result. The pseudocode provides a high-level understanding of the algorithm’s steps.

# Optimality of Algorithm and Time Complexity

Here we continue to review the optimality and time complexity of the algorithm. The algorithm's greedy approach involves sorting the files by size and then placing each file onto the disk with the lowest available storage that can accommodate it. This approach often leads to good solutions by maximizing space utilization (Kuhnle, A. 2019). Consider a scenario where the available disks have varying storage capacities, and there exists an arrangement of files that would fill in all disks, leaving no unused storage. The algorithm may not be able to achieve this optimal solution if the order of file placement does not align with the specific disk capacities (Schuetz & Caflisch 2008). Below in Table 1, we present an example, to reflect that though this is a good option, it is not the most optimal option or in other words, the perfect option.

Table 1.

|  |  |  |  |
| --- | --- | --- | --- |
| Greedy Algorithm | | Optimal Algorithm | |
| Disc Size (d) | File Size (f) | Disc Size (d) | File Size (f) |
| 7 | 5+2 | 7 | 2+2+3 |
| 3 | 3 | 3 | ~ |
| 5 | 2 | 5 | 5 |

In this example, a comparison between a greedy algorithm and the optimal algorithm or “Perfect algorithm” is shown with the output of results after calling the functions. On the left, we can see that the disc sizes of the greedy algorithm are the same as the sizes of the Optimal algorithm and in the same order. The array to sort in this example is: “5,3,2,2”. The greedy approach is to fill in the order of values that come next, as seen in Table 1. All the disks have been used, but with additional space left on the disk with the size of 5, whereas in comparison to the disks in the optimal algorithm, there is a disk unused, all the files were sorted into the proper disk and the disk with the size of 3 can be returned for a refund.

While analyzing the time complexity of the algorithm in the pseudocode, we consider the “HeapSort” time. Heap Sort takes O(n log n) time, so we must consider this when reviewing the time analysis for the algorithm as a whole (Sharma, et al. 2008). The two loops including the outer loop iterate over “n” files and the inner loop iterates over “m” disks. In the worst case, each file might need to be compared with all disks, therefore the time complexity of the loops is O(n • m). The dominant variable in the time complexity is the forming step (O(n log n)), which is followed by the loops (O(n • m)) (Sharma, et al. 2008). Therefore the time complexity of the algorithm is O(n log n + (n • m). The performance of the algorithm will depend on the specific file and disk configurations.

# Options of Brute Force Algorithm and Time Complexity

Regarding Brute Force algorithms, the brute force or exhaustive search method would not be nearly as optimal. The time complexity would be significantly higher compared to the greedy algorithm provided (Marx & Pilipczuk, 2015). Here we will review this time complexity and why it is not an optimal choice when compared to the greedy algorithm. In a brute-force approach, we would need to explore all possible combinations of file placements on disks to determine the arrangement that minimizes wasted space. In doing this, we would ultimately create openings for an error in wasted time. This involves trying every possible mapping of files to disks, which results in a time complexity of O(nm), where “n” is the number of files and “m” is the number of disks (Marx & Pilipczuk, 2015).

Each of the “n” files can be placed onto any of the “m” disks, leading to a total of “ nm ” possible combinations, which results in O(nm). The brute force approach also explores all possible ways to allocate files to disks which results in an exponential growth in time complexity as the number of files and disks increases. As the sizes of our variables “n” and “m” grows, the number of combinations grows exponentially. This could lead to impractical computational requirements which makes the brute force approach highly inefficient and could never be practical in use for the goals we are trying to accomplish. In comparison to the greedy algorithm, the time complexity is much different. The time complexity for the greedy algorithm is easier to manage than an exponential complexity like the brute force approach. While the greedy algorithm doesn’t guarantee an optimal solution in all cases, it offers a reasonable case of efficiency and finding good solutions for many scenarios, including our endeavor to file to disk allocation.

# References

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