بسم ﷲ الرحمن الرحيم

Kingdom of Saudi Arabia Ministry of Education College of Computer



المملكة العربية السعودية

وزارة التعليم

كلية الحاسب

قسم علوم الحاسب

Department of Computer Science

Algorithm design & Analysis Project CS315 – Group:338

Semester:461

Project report

Supervisor: Prof. Dr. Mohammed Al-Hagery

Rayan altawijari 431107823

Mohaya Almutairi 431107802

Mohammed Alsharekh 431108100

Fares Alsuhaibani 431107834

Abdullah Alfayez 431108148

### Table of Contents

1. **Introduction** .................................................... 3
   * The Problem of Large-Scale File Management .......... 3
   * Proposed Solutions ................................................ 3
     + Naive Algorithm .................................................. 3
     + Optimized Algorithm ............................................ 3
   * Implementation and Testing of Algorithms .............. 3
2. **Problem Description** .......................................... 3
   * Examples ............................................................... 4
     + Example 1 .......................................................... 4
     + Example 2 .......................................................... 4
3. **Algorithms and Pseudocode** ................................ 5
   * Naive Algorithm .................................................... 5
   * Optimized Algorithm .............................................. 6
4. **Implementation Details** ....................................... 7
   * Naive Algorithm Implementation ................................ 7
   * Optimized Algorithm Implementation .......................... 9
5. **Theoretical Complexity Analysis** ............................ 11
   * Naive Algorithm Complexity ....................................... 11
   * Optimized Algorithm Complexity ................................ 11
6. **Empirical Results** ................................................ 11
   * Test Environment ..................................................... 11
   * Performance Metrics ................................................ 11
   * Results and Discussion ............................................. 11
7. **Comparison and Discussion** .......................................................... 12
   * Theoretical vs. Empirical Results................................................ 12
   * Discrepancies .................................................... 12
8. **Conclusion**......................................................... 12
9. **References** .......................................................... 13

**1. Introduction**

In the realm of large-scale file management, duplicate files often accumulate, leading to inefficient use of storage space and complicating organizational efforts. This project aims to tackle the challenge of identifying and removing duplicate files by proposing and implementing two distinct solutions:

1. **Naive Algorithm**: This approach involves a straightforward, exhaustive comparison of files. Each file is compared against every other file to check for duplicity. While this method is simple to understand and implement, it can be time-consuming and resource-intensive, especially as the number of files grows. The naive algorithm serves as a baseline against which the performance of more sophisticated methods can be measured.
2. **Optimized Algorithm**: To address the inefficiencies of the naive approach, an optimized algorithm is proposed. This method leverages advanced techniques to minimize the number of necessary comparisons and improve overall performance. The optimization might include strategies to quickly identify potential duplicates without exhaustive comparison. This algorithm aims to balance accuracy with efficiency, making it suitable for larger datasets where the naive method would be impractical.

Both algorithms are implemented in Python and subjected to rigorous testing on directories with varying file counts. The goal is to not only assess their theoretical complexities but also to evaluate their empirical performance in real-world scenarios. The results of these tests provide insights into the practicality and scalability of each approach, guiding users in selecting the most appropriate method for their needs.

This report looks into the design and implementation of the algorithms, exploring their strengths and limitations. By providing a comprehensive analysis, it seeks to offer valuable solutions to the problem of duplicate file removal in large-scale file management.

**2. Problem Description**

In many large-scale file management scenarios, it is common to encounter duplicate files, which are files that have identical content but may have different names. The presence of such duplicates can lead to wasted storage space and complicate organizational efforts. The objective of this project is to detect and delete these duplicate files from a specified folder.

* **Input:** A folder containing multiple files.
* **Output:** A list of retained files and a report of deleted duplicates.

**Example 1**

* **Input:** Folder containing text1.txt, text1-copy.txt, and text2.txt.
* **Output:** Keep text1.txt, text2.txt; Delete text1-copy.txt.

A white paper with black text

Description automatically generatedA screenshot of a computer

Description automatically generatedصورة تحتوي على نص, لقطة شاشة, رسم بياني, الخط

تم إنشاء الوصف تلقائياً

Input output delete

Figure 2.1

**Example 2**

* **Input:** Folder with image1.jpg, image2.jpg (same content).
* **Output:** Keep image1.jpg; Delete image2.jpg.

A close up of a leaf

Description automatically generated

Figure 2.2

## ****Algorithms and Pseudocode****

### **3.1 Naive Algorithm**

The naive approach compares every file against every other file. If two files are identical, one of them is deleted.

#### **Pseudocode:**

**algorithm deletdup\_1.0(folder)**

**input:** folder

**Output:** folder without the duplicated files

root\_path **<-** folder path

list\_of\_files1 **<-** list of all the files in the root\_path

list\_of\_files2 **<-** list of all the files in the root\_path

dup\_files **<-** new empty array to store duplicate files

**for each** root, folders, files **in** list\_of files1 **do**

**for** **each** file **in** files **do**

file1\_path **<-** first file path

**for each** root1, folders1, files1 **in** list\_of\_files2 **do**

**for each** file1 in files1 **do**

file2\_path **<-** second file path

**if** file1\_path **and** file2\_path are identical **and** **not** the same file **then**

**if** file1\_path is **not** already **in** dup\_files **then**

**add** file2\_path **to** dup\_files

**for each** file **in** dup\_files

**delete** the file

**Print** message that the file has been deleted

Agorithm3.1

 **Time Complexity:** O(n²), where n is the number of files in the folder.

 **Space Complexity:** O(1), as no additional data structures are used.

### **3.2 Optimized Algorithm**

The optimized algorithm puts the unique files in a dictionary and when it finds a file already there it deletes it.

#### **Pseudocode:**

**algorithm deletdup\_2.0(folder)**

**input** folder

**Output** folder without the duplicated files

file\_path **<-** folder path

list\_of\_files **<-** list of all the files in the file\_path

unique\_files **<-** empty dictionary to store the unique files

**for** **each** root, folders, files **in** list\_of files **do**

**for each** file **in** files **do**

file\_path **<-** store the file path

**open** file\_path in binary read mode

**set** file\_content **=** read all contents of the file

**if** file\_content **not** **in** unique\_files **then**

**add** file\_content as key and file\_path as value to unique\_files

**else**

**remove** file\_path from file system

**print** file\_path + " has been deleted"

Agorithm3.2

 **Time Complexity:** O(n), where n is the number of files in the folder.

 **Space Complexity:** O(n), as no additional data structures are used.

## ****Algorithms and Pseudocode****

### **4.1 Naive Code**

#### **Approach**

We will ask the user to select a folder & we will search under this directory for all the duplicate files. We will open the first file and compare it with every file in the folder. When we find a duplicated file we will add it to an array that contains all the duplicate files. After that, we will traverse through this array and delete all the files in it.

#### **Stepwise Implementation:**

first we import Tkinter, os, hashlib & pathlib libraries then we use tk.withdraw because we don't want the GUI window of tkinter to appear on our screen we only want the file dialog for selecting the folder. askdirectory(title=”Select a folder”) this line of code pop ups a dialog box on the screen through which we can select a folder. After that we need to list out all the files inside our root folder. To do that we need OS module, os.walk() takes the path of our root folder as an argument and it will walk through each subdirectory of the folder given to it and it will list out all the files This function returns a list of tuples with three elements. The first element is the path to that folder and the second element is all the subfolders inside that folder and the third element is list of all the files inside that folder. Our final goal is to list out all the files in each and every subdirectory and the main directory that’s why we are running a for loop on all the files. We will compare each file with all the files in the folder in order to find the duplicated files using a filecmp.cmp() function and add it to dup\_files array . After we find all the duplicated files we need a for loop to go through the array and delete each file.

#### **Python code:**

import filecmp  
from tkinter.filedialog import askdirectory  
import time  
from tkinter import Tk  
import os  
from pathlib import Path  
  
# We don't want the GUI window of  
# tkinter to be appearing on our screen  
Tk().withdraw()  
  
# Dialog box for selecting a folder  
root\_path = askdirectory(title="Select a folder")  
  
  
# To calculate execution time  
start\_time = time.time()  
  
  
# Listing out all the files  
# inside our root folder  
list\_of\_files = os.walk(root\_path)  
  
# In order to detect the duplicate files  
# we are going to define array  
dup\_files = []  
  
# Running a for loop to get all files  
# even those inside folders  
for root, folders, files in list\_of\_files:  
  
 # Running a for loop on all the files  
 for file in files:  
  
 # Finding complete file path for first file  
 file1\_path = Path(os.path.join(root, file))  
  
 # Running a for loop on all the files to compare with  
 for root1, folders1, files1 in os.walk(root\_path):  
  
 # Running a for loop on all the files to compare with  
 for file1 in files1:  
  
 # Finding complete file path for second file  
 file2\_path = Path(os.path.join(root1, file1))  
  
 # Compare the two files and check if  
 # they aren't the same file  
 if filecmp.cmp(file1\_path, file2\_path) and file1\_path != file2\_path:  
  
 # Check if the file is not already in dup\_files array  
 if file1\_path not in dup\_files:  
 # Add the duplicated file  
 # to the dup\_files array  
 dup\_files.append(file2\_path)  
  
  
# For loop to traverse dup\_files array  
# and delete all the files  
# and print the file has been deleted  
for file in dup\_files:  
 os.remove(file)  
 print(f"{file} has been deleted")  
  
  
end\_time = time.time()  
# Print the execution time  
print(f"Execution Time: {end\_time - start\_time} seconds")

code 4.1

### **4.2 Optimized Code**

#### **Approach**

We will ask the user to select a folder & we will search under this directory for all the duplicate files. We will take the content of each file & pass it through a function which is going to generate a unique string corresponding to a unique file. In order to detect duplicate files and then delete those files, we are going to maintain a python dictionary. Every time while inserting a new file record we will check if we are getting any duplicate entries in our dictionary. If we find any duplicate file, we will take the path of the file and delete that file.

#### **Stepwise Implementation:**

First, we import Tkinter, os, hashlib & pathlib libraries then we use tk.withdraw because we don't want the GUI window of tkinter to appear on our screen we only want the file dialog for selecting the folder. askdirectory(title=”Select a folder”) This line of code pops up a dialog box on the screen through which we can select a folder. After that, we need to list out all the files inside our root folder. To do that we need OS module, os.walk() takes the path of our root folder as an argument and it will walk through each subdirectory of the folder given to it and it will list out all the files This function returns a list of tuples with three elements. The first element is the path to that folder the second element is all the subfolders inside that folder and the third element is list of all the files inside that folder. Our final goal is to list out all the files in each and every subdirectory and the main directory that’s why we are running a for loop on all the files. We need to open up each In order to open a file we need to first have the path to it so here we are using another function in os module called .path.join(). So, we’ll say open the file using the file path in read mode. and read it as binary. In order to detect the duplicate files, we are going to define an empty dictionary. We will add elements to this dictionary and the key of each element is going to be file\_content and the value is going to be the file path. If file\_content has already been added to this unique files dictionary that means that we have found a duplicate file and we need to delete that file so we’ll simply delete that file using the os.remove() function. If it’s not there then we are going to add it to that dictionary.

#### **Pyrhon code:**

from tkinter.filedialog import askdirectory  
import time  
from tkinter import Tk  
import os  
from pathlib import Path  
  
# We don't want the GUI window of  
# tkinter to be appearing on our screen  
Tk().withdraw()  
  
# Dialog box for selecting a folder.  
file\_path = askdirectory(title="Select a folder")  
  
# To calculate execution time  
start\_time = time.time()  
  
# Listing out all the files  
# inside our root folder.  
list\_of\_files = os.walk(file\_path)  
  
# In order to detect the duplicate  
# files we are going to define an empty dictionary.  
unique\_files = dict()  
  
  
# Running a for loop to get all files  
# even those inside folders  
for root, folders, files in list\_of\_files:  
  
 # Running a for loop on all the files  
 for file in files:  
  
 # Finding complete file path  
 file\_path = Path(os.path.join(root, file))  
  
 # Open our files as binary to compare  
 file\_content = open(file\_path, 'rb').read()  
  
 # If file has already been added  
 # we'll simply delete that file  
 # otherwise add it to unique files  
 if file\_content not in unique\_files:  
 unique\_files[file\_content] = file\_path  
 else:  
 os.remove(file\_path)  
 print(f"{file\_path} has been deleted")  
  
  
end\_time = time.time()  
# Print the execution time  
print(f"Execution Time: {end\_time - start\_time} seconds")

code 4.2

**5. Theoretical Analysis**

| **Algorithm** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- |
| Naive Algorithm | O(n²) | O(1) |
| Optimized Algorithm | O(n) | O(n) |

Table 5.1

The **naive algorithm** has a time complexity of O(n^2), where n is the number of files in the folder. This is because each file is compared with every other file, resulting in a quadratic number of comparisons.

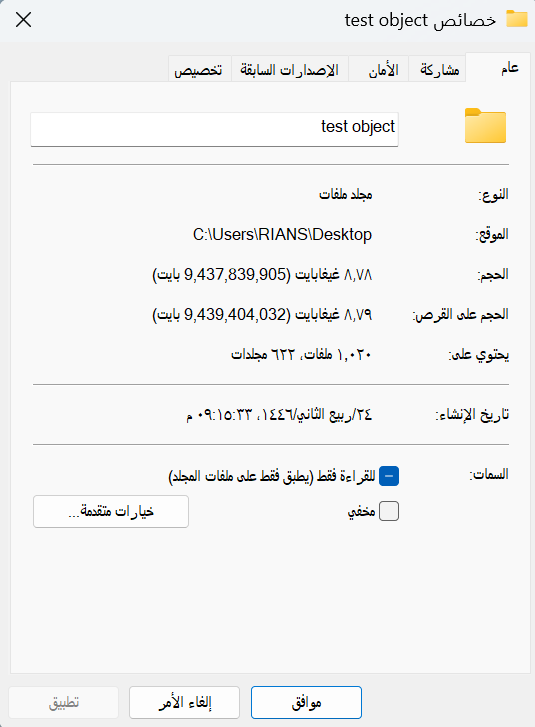
The **optimized algorithm** has a time complexity of O(n), where n is the number of files in the folder. This is because each file is processed only once to identify if it’s a unique or not

**6. Empirical Analysis**

The two algorithms were tested on folders with different numbers of files to measure execution time. Below are the results:

| **Input Size** | **Naive Algorithm Time** | **Optimized Algorithm Time** |
| --- | --- | --- |
| 10 files | 0.04seconds | 0.02 seconds |
| 100 files | 1.7 seconds | 0.08 seconds |
| 1000 files | 30.8 seconds | 0.4 seconds |

Table 6.1

The results indicate that the optimized algorithm performs significantly better as the input size increases. The naive approach is impractical for large datasets due to its quadratic time complexity. However, in this test we use the one file and copy it n times but we are going to try the algorithms in a real folder that contain many sub folders and files of deferent types we use a file that contain 1667 file (27GB)



Before After

Figure 6.1

| **Input Size** | **Naive Algorithm Time** | **Optimized Algorithm Time** |
| --- | --- | --- |
| 1667 files | 280.8 seconds | 116.3 seconds |

Table 6.2

**7. Comparison and Discussion**

* **Theoretical vs. Empirical Results:** The empirical results align with the theoretical expectations, confirming the optimized algorithm's superior scalability. The naive algorithm's performance degrades rapidly with increasing input size, while the optimized algorithm maintains a more consistent and manageable execution time.
* **Discrepancies:** Minor variations in execution times can arise from system factors like disk I/O and memory management. The optimized algorithm demonstrates significant improvements.

**8. Conclusion**

This project highlights the importance of using efficient algorithms for managing duplicate files. The naive algorithm, while straightforward, becomes inefficient for large datasets because it compares each file with every other file. The optimized approach improves this by reducing unnecessary comparisons, making it much faster and more practical for larger numbers of files. By organizing and checking files more effectively, it significantly cuts down on processing time. Our tests show that the optimized algorithm performs much better as the number of files increases. Although minor delays can occur due to system factors, the optimized method remains more efficient overall. In summary, this project emphasizes the need for thoughtful algorithm design to solve real-world problems efficiently. By refining our approach, we can handle larger datasets with better performance.

**9. References**

1. [Python Documentation](https://docs.python.org)1
2. [Python Documentation](https://docs.python.org/3/library/filecmp.html)2
3. [geeksforgeeks](https://www.geeksforgeeks.org/deleting-duplicate-files-using-python/)
4. [stackoverflow](https://stackoverflow.com/questions/748675/finding-duplicate-files-and-removing-them)
5. [wiki.python](https://wiki.python.org/moin/TimeComplexity)
6. [ChatGPT](https://chatgpt.com/)
7. [poe](https://poe.com/chat/9lsp563cijwwec376)
8. [ALGORITHM DESIGN Foundations, Analysis and Internet Examples](https://drive.google.com/file/d/1Vro56vEuYIPhrwLCxMBC3tE5parE8a08/view?usp=sharing)
9. [Introduction to Algorithms Fourth Edition](https://drive.google.com/file/d/1HRb6hn0A7qU1ypWNY9tzYzDMSDCwWK4-/view?usp=sharing)