Documentation for Algorithms Project

Team 7

Name	Id	Major
Halla Mohamed Omar	2022170858	Cyber Security
Fatma Khaled Mohamed	2022170862	Cyber Security
Karim Hossam Deghidy	2022170863	Cyber Security
Youssef Tarek Anan	2022170931	Cyber Security
Youssef Mahmoud Mohamed El-Sayed	2022170873	Cyber Security
Malak Waleed Ismail Mohamed	2022170926	Software Engineering
Renad Hossam Mohamed Ahmed	2022170898	Software Engineering

Code of the Following Algorithms & Its Analysis

a. Allocation Strategies (Heuristics)

i. Worst-fit

- **Using Linear Search :**

```
#Overall complexity for the function is O(N*M) + O(M) = O(N*M)
def WorstFit_LinearSearch(files,folder_size):         # O(1): Function definition
    folders=[]                                        # O(1): Initialize an empty list for folders
    outputlist=[]                                     # O(1): Initialize an empty list to store output
    #Overall complexity for the outer loop is N*M so it is O(N*M)
    for fileName,fileDuration in files.items():       # O(N): Loop through all files in the input dictionary where N = num
ber of audio files
        if len(folders)==0:                           # O(1): Check if the folder list is empty
            folders.append([folder_size-fileDuration,[(fileName,fileDuration)]])  # O(1):Add a new folder
        else:
            worst_fit_index=-1                         # O(1): Initialize index for the worst fit
            largest_remaining_size=-1                  # O(1): Track the largest remaining size
            #Overall complexity for the inner loop is M*1 so it is O(M)
            for i in range(len(folders)):              # O(M): Iterate over all folders where M = number of folders.
                folder = folders[i]                    # O(1): Access folder by index
                if folder[0] >= fileDuration and folder[0] > largest_remaining_size:  # O(1): Check folder capacity
                    worst_fit_index= i                  # O(1): Update worst fit index
                    largest_remaining_size = folder[0]  # O(1): Update largest remaining size

            if worst_fit_index==-1:                     # O(1): If no folder can accommodate the file
                folders.append([folder_size-fileDuration,[(fileName,fileDuration)]])  # O(1): Add new folder
            else:
                folders[worst_fit_index][0]-=fileDuration      # O(1): Update remaining space in the folder
                folders[worst_fit_index][1].append((fileName,fileDuration)) # O(1): Add file to the folder

    for folder in folders: # O(M): Loop through all folders
        outputlist.append(folder[1])  # O(1): Add folder content to the output list

    return outputlist                   # O(1): Return the final output list
```

- **Using Priority Queue:**

```
#Overall complexity of the function = O(N log M) +  O(M*log M) = O(N log M) -->Because N log M is much larger relative to M
def WorstFit_PriorityQueue(files,folder_size):          # O(1): Function definition
    folders=[]                                          # O(1): Initialize an empty list to represent the priority queue
    outputlist=[]                                       # O(1): Initialize an empty list for the output
    #Overall complexity for this loop = N * 3*log M = O(N log M)
    for fileName,fileDuration in files.items():         # O(N): Iterate through all files where N = number of audio files
        if len(folders)==0:                             # O(1): Check if priority queue is empty
            heapq.heappush(folders,[-folder_size + fileDuration,[(fileName,fileDuration)]])  #O(log M): Add folder where M =
        else:
            folder=folders[0]                           # O(1): Access the folder with the largest space
            largest_remaining_size=-folder[0]           # O(1): Convert negative size back to positive
            if largest_remaining_size >= fileDuration:  # O(1): Check if file fits in the folder
                heapq.heappop(folders)                  # O(log M): Remove folder from the priority queue
                folder[1].append((fileName,fileDuration))    # O(1): Add file to folder
                heapq.heappush(folders,[-largest_remaining_size + fileDuration,folder[1]])   # O(log M): Update folder
```

```
            else:
                heapq.heappush(folders,[-folder_size + fileDuration,[(fileName,fileDuration)]]) # O(log M): Add new folder
#Overall complexity for this loop = M * log M = O(M*log M)
    while folders:                              # O(M): Loop through all folders
        folder=heapq.heappop(folders)           # O(log M): Remove folder with the largest space
        outputlist.append(folder[1])            # O(1): Add folder content to output list

    return outputlist                           # O(1): Return the final output list
```

ii. Worst-fit decreasing

- **Using Linear Search:**

```
#Overall complexity for this function = O(max(NlogN , N*M)) + O(M) which we can ignore because it is very small = O(max(Nl
ogN , N*M))
def WorstFit_LinearSearch_Decreasing(files,folder_size):            # O(1): Function definition
    folders=[]                                                  # O(1): Initialize an empty list for folders
    outputlist=[]                                               # O(1): Initialize an empty list for output
    sorted_files=filehandler.sortduration(files)                            # O(N log N): Sort files in descending order b
y duration where N = number of audio files
#Overall comlexity for this outer loop = N * M = O(N*M)
    for fileName,fileDuration in sorted_files.items():          # O(N): Loop through sorted files
        if len(folders)==0:                                     # O(1): Check if folders list is empty
            folders.append([folder_size-fileDuration,[(fileName,fileDuration)]])   # O(1): Add new folder
        else:
            worst_fit_index=-1                                  # O(1): Initialize worst fit index
            largest_remaining_size=-1                           # O(1): Track largest remaining space
            #Overall comlexity for this inner loop = O(M)
            for i in range(len(folders)):                       # O(M): Iterate through all folders
                folder = folders[i]                             # O(1): Access folder
                if folder[0] >= fileDuration and folder[0] > largest_remaining_size:   # O(1): Check space conditions
                    worst_fit_index= i                          # O(1): Update worst fit index
                    largest_remaining_size = folder[0]          # O(1): Update largest remaining space

            if worst_fit_index==-1:                             # O(1): If no folder fits the file
                folders.append([folder_size-fileDuration,[(fileName,fileDuration)]])   # O(1): Add new folder
            else:
                folders[worst_fit_index][0]-=fileDuration       # O(1): Reduce folder space
                folders[worst_fit_index][1].append((fileName,fileDuration))           # O(1): Add file to folder
#Overall complexity for this loop = O(M)
    for folder in folders:                              # O(M): Loop through all folders
        outputlist.append(folder[1])                    # O(1): Append folder content to output

    return outputlist                                   # O(1): Return the final output list
```

- **Using Priority queue:**

```
#Overall complexity of this function = O(N log M) + O(M log M) +  O(N log N) = O(N log N) beacuse it is much greater relativ
def WorstFit_PriorityQueue_Decreasing(files,folder_size):       # O(1): Function definition
    folders=[]                                              # O(1): Initialize an empty priority queue
    outputlist=[]                                           # O(1): Initialize an empty list for output
    sorted_files=sortduration(files)                        # O(N log N): Sort files by duration  where N = number of audio
    #Overall complexity of this loop = N * 4 log M = O(N log M)
    for fileName,fileDuration in sorted_files.items():      # O(N): Iterate through sorted files
        if len(folders)==0:                                 # O(1): Check if priority queue is empty
            heapq.heappush(folders,[-folder_size + fileDuration,[(fileName,fileDuration)]])   # O(log M): Add folder
        else:
            folder=folders[0]                               # O(1): Access folder with largest space
            largest_remaining_size=-folder[0]               # O(1): Convert negative size back to positive
            if largest_remaining_size >= fileDuration:      # O(1): Check folder capacity
                heapq.heappop(folders)                      # O(log M): Remove folder
                folder[1].append((fileName,fileDuration))   # O(1): Add file
                heapq.heappush(folders,[-largest_remaining_size + fileDuration,folder[1]]) # O(log M): Update folder

            else:
                heapq.heappush(folders,[-folder_size + fileDuration,[(fileName,fileDuration)]])  # O(log M): Add new folder
#Overall complexity = O(M log M)
    while folders: # O(M) loop all folders
        folder=heapq.heappop(folders)  # O(log M): Remove folder
        outputlist.append(folder[1])   # O(1): Add folder content

    return outputlist                  # O(1): Return final output
```

iii. Sort Function & First-Fit decreasing

- **Sort-Function:**

```
def sortduration(audio):
    # Sorting the items of the dictionary by their values in descending order using Timsort
    sorted_items = sorted(audio.items(), key=lambda item: item[1], reverse=True)
    # O(N log N), where N is the number of items in the `audio` dictionary.
    # - `audio.items()` creates a view of the dictionary items → O(N).
    # - `sorted()` sorts the items using Timsort → O(N log N).
    # - The lambda function extracts the value (`item[1]`) for sorting → O(1) per call, called N times.

    # Converting the sorted list of tuples back into a dictionary
    sorted_Aura = dict(sorted_items)
    # O(N), where N is the number of items in the list `sorted_items`.
    # - `dict()` iterates over the list and constructs a new dictionary.

    return sorted_Aura  # O(1), returning the sorted dictionary.
```

- **First-Fit decreasing :**

```
def FirstFit(tracks, DDPF):
    # Sorting the tracks by duration in descending order using Timsort
    sortedtracks = filehandler.sortduration(tracks)                                  # O(N log N), Timsort complexity for sor

    Folders = []                                                                                     # O(1),
    FoldersSize = []                                                                                 # O(1),

    for key, value in sortedtracks.items():                                                          # O(N), iterating over a
        if len(Folders) == 0:                                                                        # O(1),
            Folders.append([(key, value)])                                                           # O(1),
            FoldersSize.append(value)                                                                # O(1),
        else:
            Found = False                                                                            # O(1),
            for i in range(len(Folders)):                                                            # O(M), iterating over t
                if value <= (DDPF - FoldersSize[i]):                                                 # O(1),
                    Folders[i].append((key, value))                                                  # O(1),
                    FoldersSize[i] += value                                                          # O(1),
                    Found = True                                                                     # O(1),
                    break                                                                            # O(1),
            if not Found:                                                                            # O(1),
                Folders.append([(key, value)])                                                       # O(1),
                FoldersSize.append(value)                                                            # O(1),

    return Folders                                                                                   # O(1),
    # Overall Time Complexity: O(N log N + N × M)
    # - Sorting: O(N log N)
    # - Outer loop over tracks: O(N)
    # - Inner loop over folders: O(M)
    # - Combined looping complexity: O(N × M)
```

iv. Best-Fit

- **Best-Fit Greedy:**

```
#_Code Analysis___________________________________
import heapq                                                                          #   O(1)
def best_fit(file_sizes, folder_capacity):                        #   O(1)
    # Sort files by size in descending order
    sorted_files = sorted(file_sizes.items(), key=lambda item: item[1], reverse=True)# item()-> O(1) ,sorted()->O(nlogn)"t
imsort" , total complexity->O(nlogn)

    folders = []                                                                     # O(1)

    for file_name, size in sorted_files:                                             # for loop indexing ->O(n)  , the tot
al complexity -> O(n * (m  + k)) ,Simplification: O(n * m) .
        # Skip if file is larger than folder capacity
        if size > folder_capacity:                                                   #O(1)
            raise ValueError(f"File '{file_name}' with size {size} exceeds folder capacity {folder_capacity}.") # raise Va
lueError() ->O(1)
        # Find the best folder for the current file
        best_fit_index = -1                                                          #O(1)
        min_remaining_space = float('inf')                                           #O(1)

        # Check for the folder with the smallest remaining space that can fit the file
        for i in range(len(folders)):                                               # O(m) "where m is the number of folde
rs"
            remaining_capacity, files = folders[i]                                  #O(1)
            if remaining_capacity >= size and (remaining_capacity - size) < min_remaining_space:#O(1)
                best_fit_index = i                                                  #O(1)
```

```
                min_remaining_space = remaining_capacity - size                     #O(1)

        # If no suitable folder was found, create a new one
        if best_fit_index == -1:                                                   #O(1) ,total if complexity -> O(k)
            # Create a new folder with the current file
            heapq.heappush(folders, (folder_capacity - size, [(file_name, size)])) # O(log k)  "where k is the number of f
olders in heap (having remaining capacity)".
        else:
            # Update the folder with the new file
            remaining_capacity, files = folders[best_fit_index]                    #O(1)
            files.append((file_name, size))                                        #append() -> O(1)
            folders[best_fit_index] = (remaining_capacity - size, files)           #O(1)
            heapq.heapify(folders)                                                 #O(k) "where k is the number of folders
in the heap (having remaining capacy)"
            #Rebuilding the heap to restore the heap property after modifying the folder list.

    return [folders for _, folders in folders]                                      #o(m)



#_time complexity ______________
#total code complexity is O(n·m+nlogn)
#worst case O(n^2) as m is equal to n
#best case o(nlogn) where m is equal 1

#_space complexity__________
# - The sorted_files list requires O(n) space where n is the number of files.
# - The folders list holds tuples of (remaining_capacity, files) for each folder.
#   In the worst case, this list can hold n folders, so the space complexity is O(n).
# - The heap used for managing the folders requires O(n) space because there could be up to n folders in the worst case.
# - Therefore, the total space complexity is O(n), where n is the number of files.
```

- **Best-Fit Dynamic Programming :**

```
def best_fit_dp(file_sizes, folder_capacity):                  # Total time complexity: O(n * C + n ^2)

    file_names = list(file_sizes.keys())          # O(n) - Extracting file names
    file_sizes_list = list(file_sizes.values())   # O(n) - Extracting file sizes
    n = len(file_sizes_list)                      # O(1) - Calculating number of files

    dp = [[False] * (folder_capacity + 1) for _ in range(n + 1)]
                                                  # O(n * C) - Initializing the DP table (2D list of size n+1 by C+1)
    dp[0][0] = True                               # O(1) - Base case assignment
    # Build the DP table
    for i in range(1, n + 1):                     # O(n) - Looping through files
        for cap in range(folder_capacity + 1):    # O(C) - Looping through capacities
            dp[i][cap] = dp[i - 1][cap]           # O(1) - Excluding the current file
            if cap >= file_sizes_list[i - 1]:     # O(1) - Checking if the file can fit
                if dp[i - 1][cap - file_sizes_list[i - 1]]:  # O(1) - Checking previous DP value
                    dp[i][cap] = True             # O(1) - Marking this capacity as achievable

    # Backtracking
    folders = []                                  # O(1) - Initialize list for  storing folders
    remaining_files = set(range(n))               # O(1) - Initialize set of all  files

    while remaining_files:                        # O(n) - Looping until all files are allocated
        cap = folder_capacity                     # O(1) - Reset folder capacity
        folder = []                               # O(1) - List to store current folder's files
        current_file_indices = set()              # O(1) - Set to track current folder's files

        for i in range(n, 0, -1):                # O(n) - Looping over files in reverse order
            if (i - 1) in remaining_files and dp[i][cap] and dp[i - 1][cap - file_sizes_list[i - 1]] and cap >= file_sizes_l
                                                 # O(1) - Check if file fits and is available for allocation
                folder.append((file_names[i - 1], file_sizes_list[i - 1]))  # O(1) - Add file to folder
                cap -= file_sizes_list[i - 1]    # O(1) - Update folder capacity
                current_file_indices.add(i - 1)  # O(1) - Mark file as allocated

        # Ensure progress is made
        if not folder:                          # O(1) - Check if folder is empty (no files were allocated)
            print("Warning: No more files can be allocated to a folder. Exiting.")
            break                               # O(1) - Exit if no files can be allocated

        folders.append(folder)
                                                # O(1) - Add folder to result

        # Remove the allocated files from the remaining files set
```

```
        remaining_files -= current_file_indices  # O(n) - Removing allocated files

    return folders                               # O(1) - Return the list of allocated folders


# Time complexity:
# Backtracking : O(n ^2)
# DP complexity: O(n * C)

# Space complexity:
# - DP table: O(n * C)
# - Folders and file sets: O(n)
# Total space complexity: O(n * C)
```

- **Best-Fit Linear Search:**

```
import FileHandling
import os


def look_ahead(folders, sound, folder_capacity):  #complexity: O(k)
    bestFolder = None  #complexity:O(1)
    min_remaining_space=folder_capacity #complexity:O(1)

    for i, (capacity, _) in enumerate(folders): #complexity:O(k) , where k is the number of folders
        remaining_space=folder_capacity-capacity #complexity:O(1)

        if remaining_space>=sound[1] and remaining_space<min_remaining_space: #complexity:O(1)
            bestFolder= i #complexity:O(1)
            min_remaining_space=remaining_space #complexity:O(1)

    return bestFolder #complexity:O(1)

def pack(sounds: dict[str, int], folder_capacity: int) -> list[list[tuple[str, int]]]:
    folders = []  # Complexity: O(1)
    sorted_items = sorted(sounds.items(), key=lambda item: item[1], reverse=True)  # Complexity: O(nlogn)


    for sound in sorted_items: # Complexity: O(n*k) where n is the total number of sounds
        placed = False  #comlexity:O(1)
        bestFolder=look_ahead(folders,sound,folder_capacity) #complexity:O(k)

        if bestFolder is not None: #complexity:O(1)
            capacity, content = folders[bestFolder] #complexity:O(1)
            folders[bestFolder] = (capacity + sound[1], content + [sound]) #complexity:O(1)
            placed = True #complexity:O(1)

        if not placed: #complexity:O(1)
            folders.append((sound[1], [sound])) #complexity:O(1)

    # Convert to list of lists of tuples format
    return [folder[1] for folder in folders]#complexity: O(k)


#total complexity: O(nlogn) + O(n*k) + O(k) = O(nlogn)+ O(n*k)

# Test case execution logic
if __name__ == "__main__":
    folder_capacity = 100
    packed_folders = None

    # Determine which test case to execute
    if FileHandling.workingOn_testcase == 1:
        source = r"./Sample Tests/Sample 1/INPUT/Audios"
        tracks_dict = FileHandling.t1  # Audio metadata
    elif FileHandling.workingOn_testcase == 2:
        source = r"./Sample Tests/Sample 2/INPUT/Audios"
        tracks_dict = FileHandling.t2
    elif FileHandling.workingOn_testcase == 3:
        source = r"./Sample Tests/Sample 3/INPUT/Audios"
        tracks_dict = FileHandling.t3
    else:
        source = r"./Complete Tests/Complete1/Audios"
        tracks_dict=FileHandling.t4

    try:
        audio_files = os.listdir(source)
```

```
    except FileNotFoundError:
        print(f"Directory not found: {source}")
        exit(1)

    # Process the audio metadata
    if tracks_dict:
        packed_folders = pack(tracks_dict, folder_capacity)
```

v. Next-Fit:

- **Next-Fit Divide and conquer:**

```
import heapq
def next_fit_D_C(file_sizes, folder_capacity):  # Main function to allocate files into folders.
    # Convert file_sizes dictionary into a list of tuples (filename, size)
    file_list = list(file_sizes.items())  # Converts file_sizes into a list of tuples for easier processing. O(n)

    def allocate_files(file_list):  # Recursive function to allocate files using divide-and-conquer.
        # Base case: if there's only one file, allocate it in its own folder.
        if len(file_list) == 1:
            return [[file_list[0]]]  # Single folder containing the file. O(1)

        # Divide: Split the file list into two halves.
        mid = len(file_list) // 2
        left_files = file_list[:mid]  # Left half of the file list. O(n)
        right_files = file_list[mid:]  # Right half of the file list. O(n)


        # Conquer: Recursively allocate files in the left and right halves.
        left_folders = allocate_files(left_files)  # Recursive call for left half. O(log n)
        right_folders = allocate_files(right_files)  # Recursive call for right half. O(log n)       T(N)=2T(n/2)+O(mlogm) (

        # Combine: Merge the allocated folders from both halves.
        return merge_folders(left_folders, right_folders, folder_capacity)  # Combine results. O(m log m)

    def merge_folders(left_folders, right_folders, folder_capacity):  # Function to merge folders efficiently.
        folders = left_folders  # Start with the folders from the left half. O(1)

        # Create a min-heap for folder remaining capacities.
        folders_heap = [(folder_capacity - sum(size for _, size in folder), i) for i, folder in enumerate(folders)]
        # Heap stores (remaining capacity, folder index) for each folder. O(n)

        heapq.heapify(folders_heap)  # Convert list to a min-heap. O(n)

        # Iterate through the folders in the right half.
        for folder in right_folders:  # O(m), where m is the total number of files in right_folders.
            for file_name, size in folder:  # Iterate through files in the current folder. O(k), where k is the number of fi
                placed = False  # Track whether the file is placed in an existing folder.

                while folders_heap:  # Check existing folders for available capacity. O(log n) per operation.
                    remaining_capacity, folder_index = heapq.heappop(folders_heap)  # Pop folder with the most available spa

                    if size <= remaining_capacity:  # Check if the file fits in the folder. O(1)
                        folders[folder_index].append((file_name, size))  # Add file to folder. O(1)
                        remaining_capacity -= size  # Update remaining capacity. O(1)
                        heapq.heappush(folders_heap, (remaining_capacity, folder_index))  # Push updated folder back into th
                        placed = True  # Mark the file as placed. O(1)
                        break  # Exit the loop once the file is placed. O(1)

                if not placed:  # If the file couldn't be placed in any existing folder.
                    new_folder = [(file_name, size)]  # Create a new folder for the file. O(1)
                    folders.append(new_folder)  # Add the new folder to the list of folders. O(1)
                    heapq.heappush(folders_heap, (folder_capacity - size, len(folders) - 1))  # Push new folder into the hea

        return folders  # Return the merged list of folders. O(1)

    allocated_folders = allocate_files(file_list)  # Start the recursive allocation. O(n log n)
    #Output(source,"..\Karim\k",allocated_folders,"nextfit D&C")
    return allocated_folders  # Return the final allocation of folders. O(1)


# _____________________________________Time Complexity _____________________________________#
#   1. The `allocate_files` function has a time complexity of O(n log n) because it recursively divides the file list and me
#   2. The `merge_folders` function involves iterating through the right folder list (O(m)) and inserting/removing items fro
#   3. Thus, the total time complexity of the algorithm is O(n log n + m log m), where n is the number of files and m is the
```

```
#_____________________________________Worst-Case Complexity_____________________________________#
# - The worst-case time complexity occurs when:
#   1. Every file requires its own folder because it doesn't fit into any existing folder.
#   2. The merging step involves checking all existing folders for each file, resulting in the maximum number of heap operat
# - In this case, both the recursive allocation and the merging process each take O(n log n) time, leading to a worst-case t


# _____________________________________Space Complexity_____________________________________#
# - Space used by the `file_list`: O(n) (for storing the file sizes).
# - Space used by the recursive call stack: O(log n) (due to the divide-and-conquer recursion).
# - Space used by the `folders` list: O(n) (for storing the allocated folders).
# - Space used by the min-heap `folders_heap`: O(n) (for managing the folder capacities).
# Total space complexity: O(n)
```

- **Next-Fit Greedy:**

```
def next_fit_greedy(file_sizes, folder_capacity):
    # Sort files in descending order to place larger files first
    sorted_files = sorted(file_sizes.items(), key=lambda x: x[1], reverse=True)  # O(n log n)
    # Explanation: Sorting the file sizes, where n is the number of files.

    # List to store the folders (represented as a list of tuples (file_name, size))
    folders = []  # O(1)

    # Min-heap to track the remaining capacity of folders
    heap =[]   # O(1)

    heapq.heapify(heap)

    for file_name, size in sorted_files:  # O(n)


        if heap:  # O(1)


            # Pop the folder with the largest remaining capacity
            remaining_capacity, folder_index = heapq.heappop(heap)  # O(log k)


            # If the file fits in the folder
            if remaining_capacity >= size:  # O(1)

                # Add file to the folder
                folders[folder_index].append((file_name, size))  # O(1)

                remaining_capacity -= size  # O(1)
                # Update the remaining capacity


                # Push the updated folder back into the heap
                heapq.heappush(heap, (remaining_capacity, folder_index))  # O(log k)


            else:
                # File does not fit in any existing folder, create a new folder
                new_folder_index = len(folders)  # O(1)


                folders.append([(file_name, size)])  # O(1)


                # Push the new folder's remaining capacity into the heap
                heapq.heappush(heap, (folder_capacity - size, new_folder_index))  # O(log k)

        else:  # O(1)
            # No folders yet, create the first folder
            new_folder_index = len(folders)  # O(1)

            folders.append([(file_name, size)])  # O(1)


            # Push the new folder's remaining capacity into the heap
            heapq.heappush(heap, (folder_capacity - size, new_folder_index))  # O(log k)


    return folders  # O(1)
```

```
# _____________________________________ Time Complexity_____________________________________#
# 1. Sorting files: O(n log n)
# 2. For each file (n iterations):
#    - Popping and pushing from the heap: O(log k)
# Total: O(n log n + n log k), simplified as O(n log n + n log k).

# _____________________________________ Worst-Case Complexity_____________________________________#
# The worst-case time complexity happens when:
# 1. Each file is placed in its own folder because no file fits into any existing folder.
# 2. This leads to the following operations:
#    - Sorting the files: O(n log n)
#    - For each file (n iterations), we are performing heap operations (popping and pushing):
#      - Popping and pushing from the heap: O(log n) since the maximum number of folders will be n.
# Total worst-case time complexity: O(n log n + n log n) = O(n log n).

# _____________________________________Space Complexity_____________________________________#
# 1. Storing folders: O(m), where m is the total number of folders.
# 2. Storing files: O(n), where n is the number of files.
# 3. Heap storage: O(k), where k is the maximum number of folders in the heap at any time.
# Total: O(n + m + k).
```

vi. Harmonic Partitioning:

```
import FileHandling
import os


def look_ahead(folders, sound, folder_capacity):  #complexity: O(k)
    bestFolder = None  #complexity:O(1)
    min_remaining_space=folder_capacity #complexity:O(1)

    for i, (capacity, _) in enumerate(folders): #complexity:O(k) , where k is the number of folders
        remaining_space=folder_capacity-capacity #complexity:O(1)

        if remaining_space>=sound[1] and remaining_space<min_remaining_space: #complexity:O(1)
            bestFolder= i #complexity:O(1)
            min_remaining_space=remaining_space #complexity:O(1)

    return bestFolder #complexity:O(1)

def pack(sounds: dict[str, int], folder_capacity: int) -> list[list[tuple[str, int]]]:
    folders = []  # Complexity: O(1)
    sorted_items = sorted(sounds.items(), key=lambda item: item[1], reverse=True)  # Complexity: O(nlogn)


    for sound in sorted_items: # Complexity: O(n*k) where n is the total number of sounds
        placed = False  #comlexity:O(1)
        bestFolder=look_ahead(folders,sound,folder_capacity) #complexity:O(k)

        if bestFolder is not None: #complexity:O(1)
            capacity, content = folders[bestFolder] #complexity:O(1)
            folders[bestFolder] = (capacity + sound[1], content + [sound]) #complexity:O(1)
            placed = True #complexity:O(1)

        if not placed: #complexity:O(1)
            folders.append((sound[1], [sound])) #complexity:O(1)

    # Convert to list of lists of tuples format
    return [folder[1] for folder in folders]#complexity: O(k)


#total complexity: O(nlogn) + O(n*k) + O(k) = O(nlogn)+ O(n*k)

# Test case execution logic
if __name__ == "__main__":
    folder_capacity = 100
    packed_folders = None

    # Determine which test case to execute
    if FileHandling.workingOn_testcase == 1:
        source = r"./Sample Tests/Sample 1/INPUT/Audios"
        tracks_dict = FileHandling.t1  # Audio metadata
    elif FileHandling.workingOn_testcase == 2:
        source = r"./Sample Tests/Sample 2/INPUT/Audios"
        tracks_dict = FileHandling.t2
```

```
    elif FileHandling.workingOn_testcase == 3:
        source = r"./Sample Tests/Sample 3/INPUT/Audios"
        tracks_dict = FileHandling.t3
    else:
        source = r"./Complete Tests/Complete1/Audios"
        tracks_dict=FileHandling.t4

    try:
        audio_files = os.listdir(source)
    except FileNotFoundError:
        print(f"Directory not found: {source}")
        exit(1)

    # Process the audio metadata
    if tracks_dict:
        packed_folders = pack(tracks_dict, folder_capacity)
```

vii. Fractional Packing:

```
import os
import concurrent.futures
from FileHandling import *

filehandler = FileHandlingClass()

def process_sound(name, duration_to_process): #complexity: O(1)
    """Mock process a sound file and print what would be done."""

def fractional_packing(tracks, total_duration_available): #complexity: O(nlogn) + O(n) + O(k*n)

    sortedtracks = sorted(tracks.items(), key=lambda item: item[1], reverse=True) #complexity: O(nlogn)
    # List to store packed folders
    folders = []  #complexity: O(1)
    # Tracks in the current folder
    current_folder_tracks = []  #complexity:O(1)
    current_folder_duration = 0  #complexity:O(1)

    for sound_name, duration in sortedtracks: #complexity: O(n)
        if current_folder_duration + duration <= total_duration_available: #complexity: O(1)
            current_folder_tracks.append((sound_name, duration)) #complexity: O(1)
            current_folder_duration += duration #complexity: O(1)
        else:
            # calculates the fraction of the track that can fit into the remaining folder capacity.
            fraction_to_fit = (total_duration_available - current_folder_duration) / duration #complexity: O(1)
            fraction_duration = duration * fraction_to_fit #complexity: O(1)
            current_folder_tracks.append((sound_name, fraction_duration))  #complexity: O(1)

            # Store the remaining part in a new folder
            remaining_duration = duration - fraction_duration #complexity: O(1)
            folders.append(current_folder_tracks)  #complexity: O(1)
            current_folder_tracks = [(sound_name, remaining_duration)]  #complexity: O(1)         # Start new folder
            current_folder_duration = remaining_duration #complexity: O(1)          # Reset folder duration

    # Add the last folder if it has any tracks
    if current_folder_tracks: #complexity: O(1)
        folders.append(current_folder_tracks) #complexity: O(1)

    # Calculate fractions for each folder
    split_fractions = [] #complexity: O(n)
    for folder_tracks in folders: #complexity: O(k*n) , k is the number of iterations
        for track in folder_tracks: #complexity: O(n)
            fraction = track[1] / total_duration_available  #complexity: O(1)
            split_fractions.append((track[0], fraction)) #complexity: O(1)

    # Initialize ThreadPoolExecutor with the number of CPU cores, with: ensures proper cleanup of resources after block is e
    with concurrent.futures.ThreadPoolExecutor(max_workers=os.cpu_count()) as executor: #complexity: O(n)
        #list of objects representing the execution of async tasks
        futures = [
            executor.submit(
                process_sound,
                entry[0],
                #duration to process
                entry[1] * total_duration_available,
            )
            #submit the task for each entry
            for entry in split_fractions
        ] #complexity: O(n)
```

```
        for future in concurrent.futures.as_completed(futures): #complexity: O(n)
            future.result() #complexity: O(1)

    return folders #complexity: O(1)

#total complexity: o(nlogn) + o(k*n) + o(n)= o(nlogn) + o(k*n)
#worst case: number of files= number of folders, complexity= o(nlogn) + o(n^2)= o(n^2)
```

b. Folder Filling Algorithm

```
def folder_filling(files, folder_capacity):
    # dp function returns two things:
    # 1) Maximum value obtained by including or not including the current file
    # 2) List of files used to achieve this maximum value

    def dp(names, index, remaining_duration, memo):                                               #O(n*D)
        # Base case
        if index == len(names) or remaining_duration == 0:                                        #len() is O
(1)      -> O(1)
            return 0, []                                                                          #
-> O(1)

        # Check if result is already computed and stored in memo
        if (index, remaining_duration) in memo:                                                   #
-> O(1)
            return memo[(index, remaining_duration)]                                              #
-> O(1)

        file_name = names[index]                                                                  #
-> O(1)
        file_duration = files[file_name]                                                          #
-> O(1)

        # leave_value is the maximum value obtained by not including this file
        # leave_files is the list of files used to achieve leave_value
        leave_value, leave_files = dp(names, index + 1, remaining_duration, memo)
        take_value, taken_files = 0, []

        if file_duration <= remaining_duration:                                                   #O(1)
            # take_value is the maximum value obtained by including this file
            # take_files is the list of files used to achieve take_value
            take_value, taken_files = dp(names, index + 1, remaining_duration - file_duration, memo)
            take_value += file_duration                                                           #O(1)
            taken_files = [(file_name, file_duration)] + taken_files                              #O(1)

        # Choose the better option: including or not including the current file
        if leave_value > take_value:                                                              #O(1)
            memo[(index, remaining_duration)] = (leave_value, leave_files)                        #O(1)
        else:
            memo[(index, remaining_duration)] = (take_value, taken_files)                         #O(1)
           #names.remove(file_name)                                                                #O(1)
        return memo[(index, remaining_duration)]                                                  #O(1)

    # Main logic of folder filling function
    folders = []
    files_names = list(files.keys())

    while files_names:                                                                            #O(n)
        memo = {}
        # Get the best subset of files for the current folder capacity
        _, files_in_a_folder = dp(files_names, 0, folder_capacity, memo)                          #O(n*D)
        if not files_in_a_folder:                                                                 #O(1)
            break                                                                                 #O(1)
        folders.append(files_in_a_folder)                                                         #O(1)

        # Remove the files that have been added to the current folder
        for file_name, _ in files_in_a_folder:                                                    #O(k)
            files_names.remove(file_name)                                                         #O(n)
                                                                                                 #for loop comp
lexity -> O(n*k)
    return folders                                                                                #O(1)
    #Assume that k = 1, this is the worst case scenario that at each iteration we only remove 1 file so the n is reduced s
lowly
    #if k > 1 this means that n is reduced faster. So, when k = 1 the time complexity is upper bounded by O(n)

    # Total Time complexity of while loop : (O(n*D) + O(n))*O(n) = O(n^2 * D) + O(n^2) = O(n^2 * D)
```

```
# Time complexity of dp function without memoization: T(n) = 2T(n-1) + O(1)   -> O(2^n)
# Time complexity of dp function with memoization:    O(n*D)
```

c. File Handling

```
import os
import shutil
#from traceback import print_tb

class FileHandlingClass:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super(FileHandlingClass, cls).__new__(cls, *args, **kwargs)
        return cls._instance

    def __init__(self):
        if not hasattr(self, "initialized"):
            self.initialized = True

    @staticmethod
    def hms_to_seconds(time_str):
        hours, minutes, seconds = time_str.split(':')
        return int(hours) * 3600 + int(minutes) * 60 + int(seconds)

    @staticmethod
    def seconds_to_hms(seconds):
        hours = seconds // 3600
        minutes = (seconds % 3600) // 60
        secs = seconds % 60
        return f"{hours:02}:{minutes:02}:{secs:02}"

    @staticmethod
    def readfile(folderdir):
        file_data = {}
        target_path = os.path.abspath(folderdir)
        with open(target_path, 'r') as file:
            num_entries = int(file.readline().strip())

            for i in range(num_entries):
                line = file.readline().strip()

                filename, time_str = line.split()

                key = filename

                value = FileHandlingClass.hms_to_seconds(time_str)

                file_data[key] = value

        return file_data

    @staticmethod
    def sortduration(audio):
        # Sorting the items of the dictionary by their values in descending order using Timsort
        sorted_items = sorted(audio.items(), key=lambda item: item[1], reverse=True)
        # O(N log N), where N is the number of items in the `audio` dictionary.
        # - `audio.items()` creates a view of the dictionary items → O(N).
        # - `sorted()` sorts the items using Timsort → O(N log N).
        # - The lambda function extracts the value (`item[1]`) for sorting → O(1) per call, called N times.

        # Converting the sorted list of tuples back into a dictionary
        sorted_Aura = dict(sorted_items)
        # O(N), where N is the number of items in the list `sorted_items`.
        # - `dict()` iterates over the list and constructs a new dictionary.

        return sorted_Aura  # O(1), returning the sorted dictionary.

    @staticmethod
    def Output(src, dest, folder, funcname, sample):
        # path
        outputdir = os.path.join(os.path.abspath(dest), rf"OUTPUT\{sample}", funcname)
        os.makedirs(outputdir, exist_ok=True)
```

```
        # ///////////////////////////////////////////////////////////////////////////////
        it = 1
        # Display
        print("Folders Content:")
        for i in folder:
            # path
            currentfolder = os.path.join(outputdir, f"F{it}")
            os.makedirs(currentfolder, exist_ok=True)
            # ///////////////////////////////////////////////////////////////////////////////
            # text file
            with open(os.path.join(outputdir, f"F{it}_METADATA.txt"), "w") as file:
                file.write(f"F{it}\n")
            # ///////////////////////////////////////////////////////////////////////////////
            timesum = 0
            for j in i:
                sourcefile = os.path.join(os.path.abspath(src), j[0])
                destfile = os.path.join(currentfolder, j[0])
                # txt file
                # convert sec to time format
                time = FileHandlingClass.seconds_to_hms(j[1])
                # Open the file in write mode ('w')
                with open(os.path.join(outputdir, f"F{it}_METADATA.txt"), "a") as file:
                    file.write(f"{j[0]} {time}\n")
                timesum += j[1]

                try:
                    shutil.copyfile(sourcefile, destfile)
                    # print("File copied successfully.")

                # If source and destination are same
                except shutil.SameFileError:
                    print("Source and destination represents the same file.")

                # If destination is a directory.
                except IsADirectoryError:
                    print("Destination is a directory.")

                # If there is any permission issue
                except PermissionError:
                    print("Permission denied.")

                # For other errors
                except:
                    print("Error occurred while copying file.")
            # write to text file
            with open(os.path.join(outputdir, f"F{it}_METADATA.txt"), "a") as file:
                file.write(f"{FileHandlingClass.seconds_to_hms(timesum)}\n")
            # ///////////////////////////////////////////////////////////////////////////////
            with open(os.path.join(outputdir, f"F{it}_METADATA.txt"), "r") as file:
                # Read the file's content
                content = file.read()
                # Print the file's content
                print(content)
            it += 1
        print(f"Number of folders: {it-1}")
```

Time Complexity Summary for All Algorithms

Complete Test Case 1 on PC1 :

Algorithm Type	Algorithm Details	Time Complexity	Number Of Folders	Time Of Execution
Worst-fit Linear Search	Scans folders to find the worst fit (largest remaining space) or adds a new one.	$O(N * M)$	**102**	**0.9 seconds**
Worst-fit Priority Queue	Uses a priority queue to quickly find the folder with the largest remaining space.	$O(NlogM)$	**102**	**0.96 seconds**
Worst-fit Decreasing Linear Search	Sorts files by size, then applies the linear search method.	$O(max(NlogN, N * M))$	**100**	**1.1 seconds**
Worst-fit Decreasing Priority Queue	Sorts files by size, then uses a priority queue for efficient allocation.	$O(NlogN)$	**100**	**1.9 seconds**
Harmonic Partitioning	Uses categories based on size thresholds	$O(NlogN + N * M)$	**100**	**1.75 seconds**
First-fit Linear Search Decreasing	Files sorted then search for First-Fit folder	$O(NlogN + N \times M)$	**100**	**2.02 seconds**
Best-fit With Dynamic Programming	guarantees that you will find an optimal solution	$O(n * C + n^2)$	**105**	**1.49 seconds**

Algorithm Type	Algorithm Details	Time Complexity	Number Of Folders	Time Of Execution
Best-fit With Priority Queue	Prioritizes best space utilization	$O(nlogn) + O(n*m)$	**100**	**1.7 seconds**
Fractional Packing	Handles partial file fitting	$O(nlogn) + O(n*m)$	**100**	**2.09 seconds**
Next-fit Divide and Conquer approach	Allocates the files using D&C	$O(nlogn + mlogm)$	**104**	**1.7 seconds**
Next-fit Greedy approach	Aims for a globally optimal solution	$O(n * logn + n * logk)$	**107**	**1.76 seconds**
Look-ahead Packing	Evaluates ahead to optimize packing	$O(nlogn) + O(n*m)$	**100**	**1.47 seconds**
Folder Filling Dynamic Programming	Uses memorization to maximize the used space	$O(n^2 \times D)$	**100**	**10.7 seconds**

Complete Test case 2 on PC2 :

Algorithm Type	Algorithm Details	Time Complexity	Number Of Folders	Time Of Execution
Worst-fit Linear Search	Scans folders to find the worst fit (largest remaining space) or adds a new one.	$O(N * M)$	**1046**	**8.9 seconds**
Worst-fit Priority Queue	Uses a priority queue to quickly find the folder with the largest remaining space.	$O(NlogM)$	**997**	**8.3 seconds**
Worst-fit Decreasing Linear Search	Sorts files by size, then applies the linear search method.	$O(max(NlogN, N * M))$	**1046**	**8.1 seconds**
Worst-fit Decreasing Priority Queue	Sorts files by size, then uses a priority queue for efficient allocation.	$O(NlogN)$	**997**	**7.95 seconds**
Harmonic Partitioning	Uses categories based on size thresholds	$O(NlogN + N * M)$	**997**	**8.15 seconds**
First-fit Linear Search Decreasing	Files sorted then search for First-Fit folder	$O(NlogN + N \times M)$	**997**	**7.2 seconds**
Best-fit With Dynamic Programming	guarantees that you will find an optimal solution	$O(n * C + n^2)$	**999**	**9.57 seconds**
Best-fit With Priority Queue	Prioritizes best space utilization	$O(nlogn) + O(n*m)$	**997**	**8.6 seconds**
Fractional Packing	Handles partial file fitting	$O(nlogn) + O(n*m)$	**997**	**8 seconds**
Next-fit Divide and Conquer approach	Allocates the files using D&C	$O(nlogn + mlogm)$	**1059**	**7.95 seconds**
Next-fit Greedy approach	Aims for a globally optimal solution	$O(n * logn + n * logk)$	**1078**	**8 seconds**
Look-ahead Packing	Evaluates ahead to optimize packing	$O(nlogn) + O(n*m)$	**997**	**7.9 seconds**
Folder Filling Dynamic Programming	Uses memorization to maximize the used space	$O(n^2 \times D)$	**999**	**2114.6 seconds ≈ 35.2433minutes**

Complete Test case 3 on PC3:

Algorithm Type	Algorithm Details	Time Complexity	Number Of Folders	Time Of Execution
Worst-fit Linear Search	Scans folders to find the worst fit (largest remaining space) or adds a new one.	$O(N * M)$	**10348**	**125.445 seconds**
Worst-fit Priority Queue	Uses a priority queue to quickly find the folder with the largest remaining space.	$O(NlogM)$	**10348**	**42.611 seconds**
Worst-fit Decreasing Linear Search	Sorts files by size, then applies the linear search method.	$O(max(NlogN, N * M))$	**10012**	**133.279 seconds**
Worst-fit Decreasing Priority Queue	Sorts files by size, then uses a priority queue for efficient allocation.	$O(NlogN)$	**10012**	**42.042 seconds**
Harmonic Partitioning	Uses categories based on size thresholds	$O(NlogN + N * M)$	**10154**	**100.934 seconds**
First-fit Linear Search Decreasing	Files sorted then search for First-Fit folder	$O(NlogN + N \times M)$	**10012**	**138.452 seconds**
Best-fit With Dynamic Programming	guarantees that you will find an optimal solution	$O(n * C + n^2)$	**10089**	**274.944 seconds**
Best-fit With Priority Queue	Prioritizes best space utilization	$O(nlogn) + O(n*m)$	**10008**	**514.689 seconds**
Fractional Packing	Handles partial file fitting	$O(nlogn) + O(n*m)$	**9986**	**54.905 seconds**
Next-fit Divide and Conquer approach	Allocates the files using D&C	$O(nlogn + mlogm)$	**10694**	**60.630 seconds**
Next-fit Greedy approach	Aims for a globally optimal solution	$O(n * logn + n * logk)$	**10799**	**48.952 seconds**
Look-ahead Packing	Evaluates ahead to optimize packing	$O(nlogn) + O(n*m)$	**10008**	**179.278 seconds**
Folder Filling Dynamic Programming	Uses memorization to maximize the used space	$O(n^2 \times D)$	——	**>2 hours**

Explanation of Parameters:

- **N, n**: Number of files.
- **M, m**: Number of folders (different contexts use M or m).
- **k**: Number of folders in heap (with remaining capacity)
- **D**: Total capacity in some unit, relevant in dynamic programming contexts.
- **C**: Folder capacity