**Part 1: Greedy Algorithm Design**

***Explanation***

The algorithm starts by arranging the files from largest to smallest, giving priority to the larger files to make the most efficient use of disk space. For each file, it evaluates all available disks to find the one that can accommodate the file while leaving the least amount of unused space, ensuring minimal wastage. Once the best disk for a file is found, the file is placed on that disk, and the disk's remaining storage capacity is updated to account for the new file.

**Link to Psuedocode:** <https://colab.research.google.com/drive/1Nt4ZGOQ7ke0ddQDW1Qnq8HmmFoaJKKVH?usp=drive_link>

# Define the function

def organizeFilesOnDisks(numFiles, fileSizes, numDisks, diskCapacities):

# Initialize output: track the disk assigned to each file

diskAssignments = [-1] \* numFiles # -1 indicates unassigned

# Copy the disk capacities to track remaining space

availableSpace = diskCapacities[:]

# Pair each file size with its index and sort in descending order of size

fileInfo = [(fileSizes[i], i) for i in range(numFiles)]

fileInfo.sort(key=lambda x: x[0], reverse=True)

# Iterate through the sorted files

for fileSize, fileIndex in fileInfo:

# Initialize variables to track the best fit for the current file

chosenDisk = -1

smallestWaste = float('inf')

# Check all disks to find the best fit

for diskIndex in range(numDisks):

if availableSpace[diskIndex] >= fileSize:

wasteSpace = availableSpace[diskIndex] - fileSize

if wasteSpace < smallestWaste:

chosenDisk = diskIndex

smallestWaste = wasteSpace

# Assign the file to the chosen disk if found

if chosenDisk != -1:

diskAssignments[fileIndex] = chosenDisk

availableSpace[chosenDisk] -= fileSize

# Return the final assignment of files to disks

return diskAssignments

### **Part 2:** Optimality and Time Complexity

#### Optimality:

The greedy algorithm is designed **to place each file** on the most suitable disk available at that moment, making it an efficient and straightforward approach. While it performs well in most cases, especially when file sizes vary significantly, it does have its limitations. The algorithm's local decision-making can sometimes prevent it from achieving the best possible configuration. For instance, it might struggle in situations where medium-sized files need to be evenly distributed across disks to maximize storage efficiency, as it lacks a broader perspective on the overall problem.

#### **Time Complexity:**

1. Sorting the files: Sorting *nn*n files by size takes *O(nlog⁡n)O (n \log n) O*(nlogn).
2. Assigning files to disks: For each file, we iterate through *mm*m disks, resulting in *O(n⋅m)O(n \cdot m) O*(n⋅m).
3. Overall Complexity: The algorithm runs in *O(nlog⁡n+n⋅m)O(n \log n + n \cdot m) O*(nlogn+n⋅m), which is efficient for moderate *nn*n and *mm*m.

### **Part 3: Brute-Force Complexity**

A brute-force approach would involve trying every possible way to assign *nn* files to *mm* disks and then evaluating each option to determine which configuration leaves the least unused space.

#### **Number of Assignments:**

Each file can be placed on any of the *mm* disks, leading to a total of *mnm^n* possible configurations.

#### **Time Complexity:**

1. Evaluating all possible assignments takes *O(mn)O(m^n)*O(mn).
2. **Justification:** For every configuration, we calculate the unused storage for all disks, which takes O(n)O(n)O(n)O(n)*O(n)O(n)*O(n). However, the exponential growth of *mnm^n*mn dominates.

In conclusion, while the brute-force approach guarantees the best solution by exploring every possible configuration, it becomes impractical for larger datasets because the number of possibilities grows exponentially. On the other hand, the greedy algorithm offers a more efficient and practical solution with a time complexity of *O(nlog⁡n+n⋅m)O(n \log n + n \cdot m)*. It performs well in most cases by making quick decisions based on the best immediate fit for each file. However, its focus on immediate opportunities means it doesn't always achieve the optimal solution, as it lacks a broader perspective of the overall problem. Despite this, the greedy algorithm remains a suitable choice for handling large-scale problems where computational efficiency is crucial.

*References*

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Kleinberg, J., & Tardos, É. (2006). *Algorithm Design*. Pearson.

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Levitin, A. (2011). *Introduction to the Design and Analysis of Algorithms* (3rd ed.). Pearson.

Papadimitriou, C. H., & Steiglitz, K**. (1998).** *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications.