Individual Project 2

CS627

Aidan Polivka

January 21, 2024

Part 1:

For this week's individual project, we were asked to use the greedy paradigm to develop an algorithm to fill an array of disks with an array of files. The goal is to use the fewest number of disks, and to optimize the amount of space used on the disks. Any additional disks will be returned to the store. Input into this algorithm will be an array of file names [0 – n], an array of file sizes [0 – n], an array of disk names [0 – m] and an array of disk sizes [0 – m]. The output should be a mapping of file names to the disk index. Being an object-oriented engineer, I think this would be best handled using custom objects to keep the disk names tied to their corresponding sizes, and the file names tied to their corresponding sizes, but for the sake of the assignment I decided to follow instructions as close as possible. Here is the pseudocode for this greedy algorithm:

```
/* This algorithm's purpose is to organize files onto the minimum number of disks.
 * With this pseudocode, we are making a number of assumptions:
 *  1. input array lengths for fileNames and fileSizes are the same
 *  2. input array lengths for diskNames and diskSizes are the same
 *  3. units for the fileSizes and diskSizes arrays are the same
 *  4. we are receiving valid input, without a need for error handling
 */
private Dictionary<string, int> OrganizeFiles(
    string[] fileNames, decimal[] fileSizes,
    string[] diskNames, decimal[] diskSizes)
{
    // Output dictionary of fileNames and disk indicies
    Dictionary<string, int> mappings = new Dictionary<string, int>();

    // Sort the files list by size descending
    MergeSort(fileNames, fileSizes); // O(nlogn)

    // Iterate through list of files
    for (int i = 0; i < fileNames.Length; i++) { // O(n * m)

        // Find the first disk that the file can fit onto
        for (int j = 0; j < diskNames.Length; j++) {
            if (fileSizes[i] <= diskSizes[j]) {

                // Create a mapping for the file name and disk index
                mappings.Add(fileNames[i], j);

                // Update the disk size at disk index
                diskSizes[j] -= fileSizes[i];

                // Break from this loop when the file is allocated
                break;
            }
        }
    }

    return mappings;
}
```

Part 2:

This algorithm will not always provide the optimal solution. Because of the nature of the greedy algorithm, it does not evaluate all the files at once. It evaluates each file largest to smallest and finds the first disk in the list that has space for it. Consider this scenario:

fileSizes	diskSizes
50	90
30	90
25	90
15	90

Using the greedy algorithm, disk 0 would store files 0 and 1, which would leave 10 mb unused. If the algorithm had a global context, disk 0 would store files 0, 2, and 3, maximizing the amount of space on the disk.

The time complexity of the greedy algorithm is the complexity of the sorting algorithm + the complexity of the file-to-disk mapping algorithm. Merge Sort's time complexity is O(nlogn), and the file-to-disk mapping algorithms complexity is O(number of files * number of disks). The worst-case scenario would be for all the drives to have 0 remaining space except for the last disk. So, for each file we would need to iterate through the entire list of drives. Therefore, the complexity of the file-to-disk mapping algorithm is O(n*m), and the overall time complexity is O(nlogn + nm).

The Merge Sort algorithm complexity dominates the nested for loops for files and drives, so the end time complexity is O(nlogn). Something to consider would be the possibility that the number of drives is equal to the number of input files. This is an unrealistic scenario, and out of character for the file organizing algorithm. If the possibility of this scenario were more realistic, it would be worth considering this algorithm to be $O(n^2)$, but here I think it is acceptable to use the Merge Sort complexity as the dominant factor.

Part 3:

To brute force a solution to this problem we would need to evaluate all permutations of the files (n!) for each disk (m). So, the complexity of such an algorithm would be O(mn!). This is drastically more complex than the greedy algorithm, and in large data sets of n and m it's impractical. There may be another way to optimize the space on these drives without brute force which is overly complex, or the greedy paradigm which is sub-optimal. But, the greedy paradigm provides a simple, maintainable solution that is acceptable enough for this scenario.