CS627 - Design and Analysis of Algorithms

IP3 - Greedy Algorithms

Kirsten Reid - June 5, 2019

[**Method:**](#_sq7oszuvccci) **3**

[**Analysis:**](#_1ipxplfkz09l) **4**

[**Pseudocode:**](#_tg5vs3xk8ji0) **4**

[**Brute Force:**](#_nkeir4v1zqd4) **5**

[**Time Complexity:**](#_nmsyjizaiuzq) **5**

[Results:](#_2petdm4i0j4j) 7

[**Correctness:**](#_iq0dorhwd7b5) **7**

[**Conclusion:**](#_hs164y4997dr) **8**

[**References:**](#_younkmhlofqe) **8**

# 

# Method:

Greedy algorithms are procedures used to obtain an optimal solution fast. The mechanism behind it is a set of rules that the program runs, specifically:

1. An input set (arrays, lists, graphs, etc.)
2. A selection method for choosing the “best”, or optimal, solution in a subset of the input
3. A weighted valuation system for discerning a “best” condition
4. An overall premise that the combination of the best of the subsets will warrant the best overall solution.

Usually, the object of a greedy algorithm is to *quickly* find an optimization, a minimum or a maximum condition. In terms of a disc space storage solution, the object was to minimize the amount of unused storage on each disc, AND the number of discs (this is because if the second constraint weren’t applied, we could buy a single disc with the exact size needed for each file for the “most optimal” solution).

In my head, the way to approach this problem was going through a loop:

* Sort by the largest file first to last
* Loop for every file, f, of size s, and seeing if it will fit into the first storage disc, d, of size, t.
* If so, place the file in, and move to the next.
* If not, move to the next disc, and repeat.
* Restart for the next file, start at disc 1, and repeat!

This ensures that as soon as a file, largest sizes first, find a home, the algorithm SHOULD be able to fill as much as possible, as soon as possible, with the least amount of discs.

Some formulas to describe this can be:

Files = f1, f2, .. fn

Sizes = s1, s2, .. sn

Discs = d1, d2, .. dm

Storage = t1, t2, .. tm

Our optimization condition becomes:

(i = 0-n) ∑fi\*si ≤ (j = 0-m)∑dj\*tj

Or in English:

For all combinations of files and their sizes should be less than or equal to all discs and their storage values.

In addition, (∑d\*t - ∑f\*s) / (∑d\*t) we want to be as close to 0 as possible.

We can use f and d as index spaces, and t and s as the actual weights in those spaces.

# Analysis:

To determine an actual coded method to find the appropriate greedy algorithm, I determined the major functions the code would employ:

1. Initializing two arrays, one a list of files and their size (index is file position, size is the value in the array at that position), and a list of data stores, and their storage capacities (same as above). For my manual testing purposes, I listed about 20 files, of random sizes, and 10 disc drives, of varying storage.
2. Clarify the bounds: the sum of all files and sizes should at least fit into one disc, and must not exceed the total of all discs (or else return all or none).
3. If the previous conditions are true, start with the largest file, on the largest storage, and go through one file at a time, fitting them where they can starting at the largest disc, until you reach the end of the files.
4. Per the condition “ The algorithm should return an array *map[i]* which contains the disk index of which the ith media file should be stored”, two array inputs of “fileArray[j]” and “discArray[i]”, map[i] will return where the file will be stored. So map[3] should return the disc index where the 4th file (0-3 = 4 positions) will be stored.

# Pseudocode:

The pseudocode is as follows:

// initialize files and discs

Int fileArray[100];

Int discArray[10];

// declare function

Function greedyAlgorithmDiskStorage (int fa[], int da[]) {

//count number of files, and number of discs

Int n = sizeof(fa)/sizeof(fa[0]);

Int m = sizeof(da)/sizeof(da[0]);

sortFAdecreasing; // mergesort input file array decreasing

sortDAdecreasing; // mergesort input disc array decreasing

if sum all files size > sum of all disc size

Return -1

If sum of all files < single largest disc

Return largest disc index = 0 //(since sorted)

Else

//loop through all the files in the input

//starting at the largest file and the largest disc space

For (int j = 0; j < n; j++){

For (int i = 0; i < m; i++){

Compare fa[j] to da[i]

If fa[j] < da[i]

Set map[i] = da[i];

Da[i] = da[i] - fa[j];

Else if fa[j] = da[i]

Set map[i] = da[i];

Da[i] = da[i] - fa[j];

Else if fa[j] > da[i];

Set i = i+1;

Compare fa[j] to da[i];

//recursion with the increased index.

When fa[j] <= da[i];

Set map[i] = da[i];

Da[i] = da[i] - fa[j];

# Brute Force:

Sample Data:

Fa[] = {60, 30, 25, 20, 20, 19, 19, 15, 15, 15, 14, 14, 12, 12, 10, 8, 7, 7, 7, 6, 6, 5, 5, 5, 5, 5, 5, 2, 1, 1, 1, 1}

DA[] = {100, 100, 100, 50, 50, 50, 20, 10, 10, 5}

Using Brute Force, we would have to calculate EVERY combination of 32 input files and 10 storage devices. However, you would be guaranteed to optimize your solution, because you would have seen every single possible combination.

The time complexity for this would be O(n2^n), as it would be a factorial combination (Chesney & Jahanian, 2019).

# Time Complexity:

Using sample data, we can calculate the Pseudocode complexity.

Here are our arrays:

Fa[] = {60, 30, 25, 20, 20, 19, 19, 15, 15, 15, 14, 14, 12, 12, 10, 8, 7, 7, 7, 6, 6, 5, 5, 5, 5, 5, 5, 2, 1, 1, 1, 1}

DA[] = {100, 100, 100, 50, 50, 50, 20, 10, 10, 5}

SUM Fa[] = {60 + 30 + 25… 1 + 1} = **377**

SUM Da[] = {100 + 100.. + 5} = **495**

Since SUM Fa < SUM Da, and SUM Fa != Da[0], then we will continue.

{since these are already sorted, we will skip sorting}

Our function will place each file into the largest storage first that has capacity. If it goes over, we move to the next storage:

|  |  |  |  |
| --- | --- | --- | --- |
| f | s | d | t |
| 0 | 60 | 0 | 100 |
| 1 | 30 | 1 | 100 |
| 2 | 25 | 2 | 100 |
| 3 | 20 | 3 | 50 |
| 4 | 20 | 4 | 50 |
| 5 | 19 | 5 | 50 |
| 6 | 19 | 6 | 20 |
| 7 | 15 | 7 | 10 |
| 8 | 15 | 8 | 10 |
| 9 | 15 | 9 | 5 |
| 10 | 14 |  | 495 |
| 11 | 14 |  |  |
| 12 | 12 |  |  |
| 13 | 12 |  |  |
| 14 | 10 |  |  |
| 15 | 8 |  |  |
| 16 | 7 |  |  |
| 17 | 7 |  |  |
| 18 | 7 |  |  |
| 19 | 6 |  |  |
| 20 | 6 |  |  |
| 21 | 5 |  |  |
| 22 | 5 |  |  |
| 23 | 5 |  |  |
| 24 | 5 |  |  |
| 25 | 5 |  |  |
| 26 | 5 |  |  |
| 27 | 2 |  |  |
| 28 | 1 |  |  |
| 29 | 1 |  |  |
| 30 | 1 |  |  |
| 31 | 1 |  |  |
|  | 377 |  |  |

## Results:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **d** | **t** | **n** | **n** | **n** | **n** | **n** | **n** | **n** | **n** |  |
| 0 | 100 | 60 | 30 | 10 |  |  |  |  |  | 100 |
| 1 | 100 | 25 | 20 | 20 | 19 | 12 | 2 | 1 | 1 | 100 |
| 2 | 100 | 19 | 15 | 15 | 15 | 14 | 14 | 8 |  | 100 |
| 3 | 50 | 12 | 7 | 7 | 7 | 6 | 6 | 5 |  | 50 |
| 4 | 50 | 5 | 5 | 5 | 5 | 5 | 1 | 1 |  | 27 |
| 5 | 50 |  |  |  |  |  |  |  |  | 0 |
| 6 | 20 |  |  |  |  |  |  |  |  | 0 |
| 7 | 10 |  |  |  |  |  |  |  |  | 0 |
| 8 | 10 |  |  |  |  |  |  |  |  | 0 |
| 9 | 5 |  |  |  |  |  |  |  |  | 0 |

As we can see, in this circumstance, the greedy algorithm was an optimal solution. However, this is dependent on the preliminary sorting modules that were employed, with the complexity of a merge sort being O(n log n).

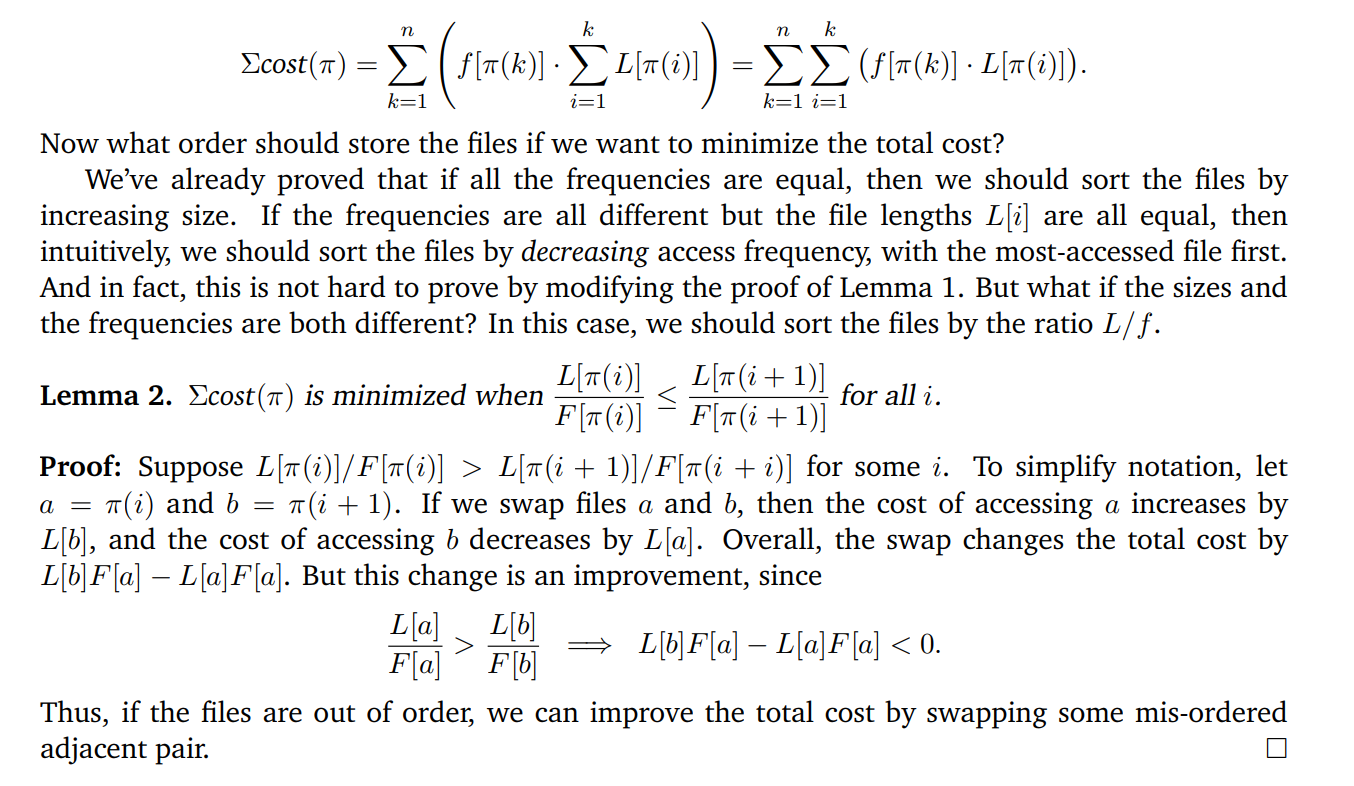
Then, having two nested for loops, one for the file loop and one for the data loop, also adds exponential time complexities of O(n^2).

If the input disc array WASN’T sorted, by chance one could get a lot of the smaller discs first, which would result in a much faster result, but with a much higher chance of being incorrect.

# Correctness:

There are many methods for determining ‘correctness’; you can count numbers of permutations, the achievement of the initial goals, but the real ‘correctness’ is revealed through the comparisons to other ‘right’ solutions. As per Roughgarden, et. all, 2013: “Exchange arguments are a powerful and versatile technique for proving optimality of greedy algorithms.” If you can prove that comparing your optimal solution to the other possible solutions is **not** equal, then you can prove your solution is (or is not) the **most** optimal.

For example, in a case for file access sizes on a tape as per Erikson, 2006:



In this instance, the pre-sorting of the disk and file sizes was a **conditional** reason why the greedy algorithm worked - if the file sizes and disk sizes were random, and the pairing was based on large files and chance disk pairings, the greedy algorithm would have failed and introduced many small discs into the solution.

One simple method to make this implementation of the greedy algorithm wrong is to change one of the lower disc sizes from 100, 100, 50, 50, 50, 20...to 100, 100, 50, 50, 50, **30** would have 27/50 spaces used up which would cost more than the 30 storage disk. Alternatively, if there were a skewing of higher values to lower ones, the ability to properly fill the disks would not be optimized going one by one, but rather matching the disparity to the next indexed file.

For example, if disc 1 has 10 free mb, disc 2 has 20 free mb, and disc 3 has 15 free mb, a 15 mb file will go into disc 2 when it could have perfectly completed the third disc.

# Conclusion:

Finding correctness is an objective science. There are definite solutions which are the *fastest*, and there are solutions which satisfy the given conditions, but no 100% way to ensure you will get the fastest, surest method (for this specific type of problem). The tradeoff will come in the application requirements and hardware limitations imposed during the investigation analysis.

# 

# References:

1. Brilliant (n.d.) *Sorting Algorithms*. Retrieved from [https://brilliant.org/wiki/sorting-algorithms](https://brilliant.org/wiki/sorting-algorithms/)
2. Geeks for Geeks (n.d.) *Analysis of Algorithms*. Retrieved from <https://www.geeksforgeeks.org/analysis-of-algorithms-set-4-analysis-of-loops/>
3. Chesney, D., & Jahanian, F. (2004) *Brute Force Algorithms and Greedy Algorithms*. University of Michigan. EECS 281. Retrieved from <https://www.eecs.umich.edu/courses/eecs281/f04/lecnotes/25-greedybrute.pdf>
4. Roughgarden, T., Sharp, A., & Wexler, T. (2013, Jul 29). *Guide to Greedy Algorithms*. Stanford. Retrieved from <https://web.stanford.edu/class/archive/cs/cs161/cs161.1138/handouts/120%20Guide%20to%20Greedy%20Algorithms.pdf>
5. Erikson, J (2006) *Greedy Algorithms*. Retrieived from <https://cs.slu.edu/~echambe5/spring09/cs314/schedule/x01-greedy.pdf>