Athens University of Economics and Business
Department: Computer Science
Course: Data Structures
Academic Year: 2019-2020
Instructor: Vangelis Markakis

2nd Project

Sorting and Priority Queues

The goal of this assignment is to familiarize you with sorting algorithms and priority queues.

This project addresses issues encountered in optimization problems regarding the allocation of memory resources for storing large volumes of data. Suppose you want to place all contents of a set of N folders and files on hard disks with a capacity of a 1 TB each (consider, for example, creating backups to a file management system). Assume that all folders are between 0 and 1,000,000 MB (1 TB) in size. The restriction we have is that each folder must be saved entirely on one disk. Ideally, the best solution would be to use as few hard drives as possible. This problem is an example of the well-known "Bin Packing" problem, for which we do not yet know if there is an efficient algorithm that can always find the best solution. However, we can design efficient methods of approaching the optimal solution.

Part A [15 points]. Before moving on to the algorithms you will implement, let's start with the Abstract Data Types you will need.

Disk ADT: To implement the algorithm you must first implement a data type that represents a 1TB size disk. Name this class Disk. Disk objects must:

- have unique ids assigned when a new disk is created (useful for debugging).
- contain a list that is named folder in which folders that are stored on this disk will be inserted during the storage algorithm.
- have the getFreeSpace() method which returns free disk space in MB.

In the Disk class you can also add any other field that you find useful. This class should implement the Comparable <Disk> interface so that you can use it in a priority queue. To implement the Comparable <Disk> interface (and therefore the compareTo function) you can simply compare the free space of each disk: if disk A has more free space than disk B then consider that A>B and the operation

A.compareTo(B)

must return the value of 1. In a similar way, when A<B the function returns -1 and when A == B the function returns 0.

Priority Queue ADT. You will need an efficient data structure that will represent a priority queue. You can either use the queue that was presented in the lab or make your own based on lecture slides. In any case, your queue should definitely have the insert and getmax (maximum key element removal) functions. Name this class **MaxPQ.**

Regarding the folders list, you are not allowed to use built in implementations of list structures from the Java library (eg ArrayList, LinkedList, etc.). You can either use the lab list or your own.

Part B [40 points]. Implement the following storage algorithm. Name your program **Greedy.java.**

1st Algorithm - Greedy: Process the folders one by one in the order in which they appear. If a folder fits on one of the disks you have already used so far, save it to the disk with the most free space. Otherwise, if it does not fit on anyone, use a new disk and save the file there.

Example: Suppose the algorithm sees 5 folders in a row sized (in MB) 200,000, 150,000, 700,000, 800,000, 100,000. The algorithm will use 3 disks. It will store folders 1,2 and 5 with capacities {200,000, 150,000, 100,000} on the 1st disk, folder 3 with capacity {700,000} on the 2nd disk (because at the time this folder was processed, it did not fit on the 1st disk), and the folder 4 with capacity {800,000} on the 3rd disk. **Note** that in this example, the best solution would be to use 2 disks instead of 3 (folders 1,4 stored on 1st disk and folders 2,3,5 stored on 2nd disk). Although this algorithm is not always optimal in terms of the number of disks used, it is a fairly fast algorithm than in many cases can approach the optimal solution.

Input and output

Input. Greedy.java program will contain a main method which will first read the file sizes from a txt file in order to be able to execute the algorithm. Each line of the file will represent the size of a folder in MB, so needs to be an integer between 0 and 1,000,000. In the example mentioned above, the input file will have the following format:

```
200000
150000
700000
800000
100000
```

If a folder does not have a size between 0 and 1,000,000 you must print an appropriate error message and terminate the program.

Output. The program should calculate and print the number of disks used by the algorithm, as well as the sum of the sizes of all folders in TB (note that this is a lower limit on the minimum number of disks required). Also, if the number of folders to be stored is not more than 100, print the contents of the disks in descending order of the empty space of each disk. For each disk print its id, the size of its free space and then the size of each folder stored on the disk. The following is an example of the output format:

```
% java Greedy input.txt
Sum of all folders = 6.580996 TB
Total number of disks used = 8
id 5 325754: 347661 326585
id 0 227744: 420713 351543
id 7 224009: 383972 392019
id 4 190811: 324387 484802
id 6 142051: 340190 263485 254274
id 3 116563: 347560 204065 331812
id 2 109806: 396295 230973 262926
id 1 82266: 311011 286309 320414
```

The example figured above shows that 8 disks were used to store data with a total size of 6.580996 TB, and e.g. disk with id 5 has 325754 MB of free space and contains 2 folders with storages 347661 and 326585 respectively.

Part C [15 points]: In the example of the 2nd page we saw that the solution of the **Greedy** algorithm was not optimal. One reason for this was that the algorithm first processed 2 small folders and that had as a result that the larger folders could not fit together with the smaller ones and required new disks. In this regard, one idea for improving the algorithm is to look at the folders in descending order from larger to smaller (in size). In this particular example if we did this, we would notice that woe would eventually put the 800 and 150 GB folders on the first disk, and the rest on the second disk. Therefore we would find a better solution using only 2 disks. So the new algorithm is as follows :

2nd Algorithm (Greedy-decreasing)

Arrange the folders in descending order and then apply the first algorithm.

To implement the 2nd Algorithm, all you need to do is first run an integer sorting algorithm in descending order. Therefore, in Part C it is required to write a program called Sort.java, which will run one of the Mergesort, Quicksort or Heapsort methods. You are free to choose whichever of the 3 methods you want and you could either use a table or a list. The main method does not need to be in Sort.java.

Attention: You are not allowed to use built in sorting functions of the Java library. You will have to implement your own method.

Part D [20 points]. In this part you will do a little experimental evaluation to determine which algorithm is best in practice. Use appropriately the Random class of the java.util package and randomly generate input data for at least 3 different values of the number of folders. Indicatively, if N is the number of folders, you can use N = 100, 500, 1000. For each value of N, create 10 different txt input files for algorithms 1 and 2. In total, you will create at least 30 random test data files. In these files, each line must be a random integer in the range [0, 1000000].

Then write a program that will execute and compare the two algorithms. For each input file, keep track of how many hard disks the two algorithms require. Calculate for each value of N, the amount of disks the two algorithms require on average, based on your random data. Part D's output must be the code you wrote to generate the input files and to compare the 2 algorithms. There is no restriction on how you will name java files.

Optional: You can use more values for N, and more than 10 files for each N, and pass the results to a spreadsheet (Microsoft Excel, OpenOffice, etc.). Use the spreadsheet capabilities to draw a diagram showing the "Number of disks as a function of the number of folders (N)". The same diagram will show two curves: one for each algorithm.

Hints and additional implementation instructions for Parts B, C and D:

- Name all classes exactly as it was requested (Disk.java, MaxPQ.java, Greedy.java, Sort.java etc).
- The implementation of the 1st Algorithm must make use of the MaxPQ priority queue. The problem can be solved without the tail, but with worse complexity. The task of this assignment is to take advantage of the priority queue.
- If you want, you can use a priority queue with generics.
- In order your code is well organized, try to divide your programs into modules according to their function. Indicatively:
  - Write a separate method that reads the input file and saves it to a table or a list (if you use tables instead of lists, you can first cound the number of lines in the input file so that you know what size is required to store the data).
  - You can implement the 1st Algorithm for the table or list you read, so that you do not have to deal with the txt input file further.
  - In this way, the implementation of the 2nd Algorithm requires only the table/list sorting and you can reuse the previous parts of our code from Part B.
- To read the input file you can use built in methods for reading files from the Java library. In addition, you can use in your code, basic data structures you have seen so far in your class or in the 1st assignment. You are not allowed to use built in implementations of list, stack, queue structures from the Java library.
- You must specify the entire path for the .txt input file in Part B, e.g. if you run it from the command line, the execution will be as follows:

  - *java Greedy path_to_file/filename*

Part E – Project Report [10 points]. Prepare a short report in a pdf file (Do not submit Word or txt files) that is named project2-report.pdf. In this report you will:

- Part A: Briefly explain how you implemented the priority queue (if you made your own or adapted the one from the lab, if you used generics, etc.). Half page at most.
- Part B: Comment on how you used the priority queue to implement the 1st Algorithm. Also comment on any other point in your code that you find useful. 1 page at most.
- Part C: Explain which sorting algorithm you implemented. Half page at most.
- Part D: Explain how random input data was generated (how you used the Random class). Also, explain what you did to run the 2 algorithms on the same input files. Next, comment on your findings and show the average for the number of disks needed for each different N value in each algorithm. Mention the conclusions you draw for the performance of the 2 algorithms. 4 pages at most.

The total size of the report should be at least 3 pages and at most 6 pages.

Additional Notes

Your program should not have any syntax errors so that it can be compiled. Programs that can't be compiled lose 50%.

Project will consist of:

1. The source code. Place the java files in a folder named **src.** Name all classes as it was requested. Also, be sure to include any other source code files needed for a successful compilation and to add explanatory comments where you think it is necessary.
2. The test data you created. Place them in a subdirectory named **data.**
3. The report.