Copenhagen Business Academy

Weekly 5

Algorithms & Data Structures

Out-of-core sorting

Authors:
Alexander Nielsen - cph-an178
Tjalfe Møller - cph-tm166
David Carl - cph-dc62

May 20, 2019

1 Introduction

We choose the 'Out-of-core sorting' assignment because it sounded like a fun challenge that involved multiple ways to attack it.

The goal of the assignment is being able to sort a file that's bigger than the amount of memory your PC has. This required us to think a bit outside the box since modern computers normally have a decent amount of memory and the files we usually have as text format isn't so big that its a problem.

2 Problem at hand

After looking over the assignment a couple of times and figuring out what needed to be done, we found some of the bigger tasks in it. The biggest problems was the following 2 things as we viewed it:

- Splitting the original file without reaching memory limit
- Merge the smaller files back into a full file while sorting it

3 Splitting files

As said in chapter 2, we need to find a way to split a file into smaller files.

There could have been several approaches into how this could be done. One of the leading ideas would be that we looped over the big file and read it line by line and add them to a Array List. We would then keep checking if our desired file size had been reached in lines (the smaller files).
And that's when we would save it to the new parts file. We found this approach ugly but doable if we weren't able to figure anything else out.

What we ended up with was a much cleaner and optimized way to do it. First we defined how big we wanted our separate files to be, this was done with a parameter to our method. We then calculated how many files we needed based on our main files size and our desired output file size.

We found out about **RandomAccessFile** which did exactly what we looked for. Instead of loading the entire file into our memory (Which is limited, hence this project) it uses a courser to set its position relative to the file byte. We could then say from byte X we want Y amount of bytes saved into another file. We used the following source code (Version 3, channels) for this. However when doing it this way, we had some problems with the files it gave us. Since we now split on Y amount of bytes sometimes it would split in the middle of our float. This caused data corruption and wouldn't work in the end, so we were back at square one. Instead of using another approach since this was

perfect for our need, we decided to find a solution to this problem. We came up with the following solutions:

- Making sure each line was Z amount of bytes, that way we could 'control' the splits
- Moving our cursor around and finding a newline byte and split there instead

We decided to give it a try to move the cursor instead of giving each line a fixed byte length since this would be a much cleaner and more versatile approach to it.

After some modification and looking into how RandomAccessFile worked we finally had a solution to our problem, where it would move the cursor back until it found a newline byte (which is 10). This was performed with a 'fori' loop that counted down instead of up (since we didn't wanna increase our file size to drastically).

With this approach our file splitter ended up having a complexity of $O(n^2)$ because we wanted to make sure we only split on newlines.

4 Sorting

For sorting the split files we choose Merge Sort, due to it's consistent and low compute time of $O(nlogn)$. The one down sight of the Merge Sort algorithm are the overhead in memory consumption. When we choose this algorithm we had forgotten that Merge Sort uses $O(n)$ memory.
To combat this we could have chosen another sorting algorithm like, Heapsort, this would have the same time-complexity as Merge Sort be a in-place sorting and therefore not consume extra memory, but it's also a unstable sorting algorithm - which in our case of float values would not matter.

5 Merging files back together

We approached our second bigger problem by this point, we managed to split out the files and sort the different files. However we wanted to merge the files back together so we got 1 big file that's sorted instead of many small files that's only sorted compared to them self.

This could be achieved with something called k-way merge algorithm. The algorithm is made for merging arrays that's already sorted together, so we had to modify it a tiny bit to work with our files instead of arrays. And since we have to keep our memory constraint in mind, we can't just load the files into arrays. So we ended up loading the first line of each file and compared them to each other, and saved down the lowest value into a new file. We then took the a new value from the same file which the lowest value was taken from originally. This way we managed to build back up a new file that had all the values that it first had. This gave us the ability of not reaching a high memory usage as we compared the different files and merged them together.

Time complexity in a k-way merge sort is $O(nlogk)$, and the space complexity is $O(n)$

6 Benchmarks

We defined our smaller files to be 25 mb.

Time for splitFile method

Size	Time	No. files
100 mb	0.089s	5
200 mb	0.110s	9
400 mb	0.197s	17
800 mb	0.489s	33
1600 mb	1.160s	65

Time for sortFiles method

Size	Time	No. files
100 mb	11.947s	5
200 mb	21.305s	9
400 mb	43.567s	17
800 mb	88.403s	33
1600 mb	186.049s	65

If we compare the time used on our splitFile method, this seems a bit slow. This could be optimised by running in threads since we are using multiple

files, and the files are small. However the amount of threads are also limited by the limiting factor in the assignment, the memory amount. So it might not be a feasible way of optimising this.

Time for mergeFiles method

Size	Time	No. files
100 mb	3.453s	5
200 mb	7.650s	9
400 mb	16.365s	17
800 mb	39.440s	33
1600 mb	67.127s	65

7 Conclusion

We found this assignment easier than first imagined, however we still learned a couple of things about memory management and IO management, things we first thought about now when writing this report so we sadly haven't implemented but also things we had a chance to implement. Since IO is time consuming we thought about having a buffer of floats in our method to merge our smaller sorted files into a bigger one, this could have achieved lower timings (faster speeds) since it would have cut down big time on IO. This could be for every time our arraylist was 1000 floats big, already now we have 1000 times less IO usage than before.

Overall it was a fun project, we learned a couple of things but also did a couple of mistakes that we could learn from.