**ANALYSIS**

1. The program first generates 2 files with 64 bit random numbers. The size of files is around 1GB. The number of numbers and the minimum and maximum values are configurable from Util.java class.

2. The program by default assumes 128 MB of free memory to find duplicates in 2 files of 1 GB each.

3. The program divides each file into a number of chunks depending upon the size of generated file and amount of memory available. The program uses BufferedReader to read the large file line by line as memory is limited and loading the whole file in memory is not possible.

4. Since each chunk fits into memory it is sorted using Java 8 parallel streams. The time complexity of sorting using Collections.sort() is O(NlogN) . Parallel stream is used for sorting to fasten the sorting of each chunk. The chunks are stored on disk for analyzing.

5. A class named BufferedReaderPro which extends BufferedReader has been created with some extra funcitonalities. Now the program uses a minimum heap of bufferedReaderPro. The chunks are then merged into a single file. **The process is exactly same as merging K sorted lists.**

BufferedWriter is used to append one number to the file in each step. One sorted file is created and the duplicates in the file are removed in the process itself. The time complexity is O(logN) for each poll method call to PriorityQueue and all the numbers are polled once. So total time complexity is O(NlogN) for this process.

6. Steps 3,4,5 are done for both the files separately.

7. Now we have 2 sorted files each of which doesn’t have any duplicates within itself. The duplicates in the file itself has been removed to make the files smaller in size.

8.Now we have to merge 2 sorted files which can be large and not fit in memory. So we use BufferedReader which can read one line at a time from the file. We use one BufferedReader for each file.

9. Now the algorithm is simple:

The in memory algorithm would look like following :

ArrayList<Integer> list1, list2; // Both are sorted.

ArrayList<Integer> duplicates = new ArrayList<>();

int i=0, j=0;

while(i < list1.size() and j<list2.size())

{

if(list[i] == list[j]) //Duplicate found

{

duplicates.add(list[i])

i++;

j++;

}

else if(list[i] < list[j])

{

i++;

}

else

{

j++;

}

}

// Now duplicates contains all the numbers which are common to both the lists.

10. The above algorithm is very similar to merge step of merge sort.

The time complexity is O(N + M) for above algorithm where N and M are sizes of the list1 and list2.

The space complexity is O(1) as we are not using any extra space for the algorithm. Duplicates array is to store the answer and hence now counted while calculating the space complexity of the algorithm.

11. We use the same algorithm as above in the case of large files. Since we can’t read the files into memory, we use BufferedReader which can read one line at a time from the file and thus serves the same purpose as I and J from above.

12. The duplicates are finally stores in duplicates.txt.

EXTENSIONS

The project can be extended to include the following. I will be happy to discuss them later in interviews.

1. There is a lot of scope to use concurrency to save time but since the problem statement didn’t say anything about using concurrency, it has not been used.

2. Writing unit tests. Again not mentioned in problem statement, hence not written. But I made sure to test the algorithm on a lot of different cases so I am quite sure that it is working correctly.

3. This was supposed to be a small assignment as per the email, but it is not so small at all. There are a lot of corner cases to cover which took some time for me to implement.

As a result, I haven’t spent a lot of time on writing very clean code. At work I take much more care when writing code.

I think this was meant to be an algorithmic assignment to check coding skills of the candidate so I took care of using most efficient algorithms available for the process.

For eg. I am not using any extra space to detect the duplicates. One other solution which is not as efficient involves using O(N + M) space for HashMap.