**Question 1.**

Similar to the Fibonacci sequence, there exists sequences that rely on previous values to calculate the current value. For example, the Fibonacci sequence contains the following numbers (0,1,1,2,3,5,8,13, …, ∞). This sequence calculates the current number by adding the previous two numbers. We can abstract this to the following two cases 𝐹𝐹0 = 0, 𝐹𝐹1 = 1 and 𝐹𝐹𝑛𝑛 = 𝐹𝐹𝑛𝑛−1 + 𝐹𝐹𝑛𝑛−2, and we assume the first two numbers are 0 and 1. We would have seen this discussed in class.

Now consider a similar sequence, where 𝐶𝐶0 = 0, 𝐶𝐶1 = 1, 𝐶𝐶𝑛𝑛 = 𝐶𝐶𝑛𝑛−1 + (2 ∗ 𝐶𝐶𝑛𝑛−2) if n is even, and 𝐶𝐶𝑛𝑛 = 𝐶𝐶𝑛𝑛−1 ∗ 𝐶𝐶𝑛𝑛−2 if n is odd (the operator \* is being used to indicate multiplication), and we assume the first two numbers are 0 and 1. Let’s call this the Clifford sequence.

1. Write an iterative solution (complete java application) for calculating all the numbers between 1 to 1000 for n in the Clifford sequence.
2. Write a recursive method (complete java application) to calculate this sequence.
3. What is the time complexity (Big O) in both cases, recursive and iterative? Explain your answer.
4. Write an algorithm (complete java application) to find the numbers that are members of both the Fibonacci sequence and the Clifford sequence. The numbers should only occur once in this series and the current number is always greater than the previous. (𝐹𝐹𝐹𝐹𝑛𝑛 > 𝐹𝐹𝐹𝐹𝑛𝑛−1).

**Question 2**

Using the same file, you worked on for your CA (staff.csv, stock.csv, or films.csv) complete the following tasks. A java application should be written to complete the following tasks.

1. Write your own solution for reading the csv into memory using the buffered reader (note the original way to read this in the provided code was using the Scanner class).
2. Write a sorting method for all possible columns in the data. Your application should allow the user the option to sort the array using any column (This can be done through either the Scanner class or using the swing windowing library).
3. Create a multi-threaded solution, where each thread sorts a copy of the data in a different order depending on the column being sorted, i.e., thread 1: sorts based on column 1, thread 2: sorts based on column 2, and so on. A copy of each ordering should be saved to a different file. For example, if sorting the films file based on column 1 save it as sortedFilms\_C1.csv (it doesn’t have to be csv, it can be whatever file format you choose).

**Question 3**

1. Discuss in your own words the following with respect to multi-threading. Locks, deadlock, starvation, and synchronization. Provide example of when each may be appropriate or can occur (i.e., when to use locks and synchronization, and what causes deadlocks and starvation).

Lock: Ao criarmos um lock damos ao objeto “com lock” uma continua execução de todo o seu código sem interferência nos dados por outra thread. Desta forma, enquanto uma thread possuir o lock, nenhum dado irá se perder ou ser modificado de maneira inesperada.

Exemplo:

Você deseja imprimir todos os dados de um array sem ser interrompido por outra thread que também está tentando imprimir, podemos adicionar o lock e fazer as outras threads esperarem o lock ser liberado.

Synchronization: Bem similar ao lock, podemos sincronizar o objeto atual (palavra-chave this) com outros métodos, assim garantimos que a chamada irá até lá terá conservado o lock iniciado na thread atual.

Exemplo: Também podemos usar a syncronization para resolver os problemas com impressão de arrays, desta forma não seremos interrompidos. No problema anterior, caso fossemos processar em threads e apresentar no console, os dados poderiam sair embaralhados visualmente, mesmo que ordenados na memória. Para garantir um print ordenado, é necessário sincronizar o método responsável pelo print.

Deadlock: No deadlock temos uma situação em que o lock é criado, usado, e nunca liberado para que outras threads também possam utilizá-lo.

Exemplo: uma situação no qual temos uma dependência de informações entre duas threads, no qual uma precisa outra e a outra precisa da anterior, e se a informação não for passada, entramos num loop com o lock ativo dentro delas.

Starvation: é também um modelo no qual não alcançamos o lock em certas threas, mas diferente da deadlock, não há nenhum impedimento de código para isso aconteça.

Exemplo: A starvation é ligada prioridade informada para as threads e se a velocidade de consumo das threads for menor do que a velocidade que elas são geradas, as threads de prioridade baixa nunca serão executadas e podem até gerar um overflow no buffer.

1. What are possible solutions to deadlock and starvation?

Deadlock solution:

não criar uma dependência entre as threads ou tratar a sua *thread pool*, para nunca criarmos uma dependência em loop, e caso isto aconteça possamos eliminar esse processo e passar o liberar o lock para o restante das threads.

Starvation solution:

Adicionar um fator idade na solução, para que a *thread pool* seja influenciada não só por um critério de ordenação, mas pelo tempo em que uma determinada thread esteve la dentro.

1. Using the producer/consumer solution from class provided on Moodle (locks already included). Include in the java application a mechanism to prioritize consumers with an ID of a higher value, i.e., if multiple consumers are waiting on a resource given by a producer ensure the consumer with a higher ID gets the resource first.
2. For the solution to part (c) now add a fairness mechanism to prevent starvation of consumers with lower ID’s while also still taking their ID into consideration.