CHALLENGE SUBMISSION

The repository consists of two main files “Transaction.h” and “BitCoin.cpp”.

Transaction.h

I have first created a class named “Transaction” to represent a transaction.

It has private members as follows:

1. tx_id = Transaction id of type *string.*
2. fee = Transaction fee of type *integer.*
3. weight = Transaction weight of type *integer.*
4. parentTransactions = A *vector* of type *string* consisting of ids of parent transactions.

Four accessor functions are created as public to get the transaction values (read only).

BitCoin.cpp

The main idea of solving the given problem was to maximize the total fees that the miner can generate by including the transactions in the block keeping the total weight below the maximum weight assigned (4000000). So, there are two cases: -

1. Either we keep on including the transactions and their parents in order of decreasing fees such that we first include the transaction with maximum fees, making sure that we first include it’s parent transactions.

2. Second case can be we keep on including the transactions in increasing order of their weight so that we start from the minimum weight and we keep on adding till the total weight reaches the limit. This way we ensure to include a greater number of transactions.

The final result would be the **maximum of the two outputs** generated by the above.

Implemnetation:

I divided the whole code into the following functions: -

1. **compareTransactionMaxFee and compareTransactionMinWeight:**
   So these are the comparator functions made to compare two transaction while sorting them in either of the two ways i.e. to get the maximum fee first and to get the minimum weight first.

2. **transactionFileParser:**
   A Helper function to read from the "mempool.csv" file and store the data in form of *vector of Transaction* objects.

3. **writeTransactionsToFile:**
   Another helper function to write the output to the "block.txt" file.

4. **sortTransactions:**
   This sorts the *vector of Transactions* in both the ways described above in single iteration. It is an implementation of a **bubble sort** (Time complexity of $O(n^2)$), which can be improved if we use an implementation of a **merge sort**.

5. **getBlockData:**
   This is the main function which has the logic of including the transactions in the block. So what I have done here is I have created an **unordered_map** to map the transaction ids of each object to it's index in the vector so that I can access parent Transactions of a particular Object in the vector in constant time. Also I am maintaining an **unordered_set** to keep a track of the transaction that are already included in the block which will allow me to check if any transaction is already included or not again in constant time. And I am pushing the transactions to a vector which will be my output. Now for each transaction I am including it in the block if it doesn't have any parent Transactions that are pending and adding it's weight to would not cross the limit and updating the total weight and total fees.
   Also if any transaction has any pending parent transactions I would create a stack and push the child transaction and keep on pushing the

pending parent transactions. Then I would get to the top of the stack and check if this transaction can be included in block or not on the basis of the same logic discussed above and if yes then I would pop it from the stack and include it in my block and would keep on doing this till the stack gets empty.

Now I am doing this for both the data which are sorted in increasing order of fees and the other one which is sorted in decreasing order of weights and which of **the two yields the maximum fees**, that will be our output.

The average time complexity calculated would be somewhere around $O(n^2)$.