Question 1:

I created three different functions for Question 1 and implemented a Stack as the data structure. Firstly, I implemented a recursive linear time algorithm for Fibonacci sequence. This reduced the run time complexity. The kth_Lucas function uses the recursive linear Fibonacci sequence algorithm in order to find L_n. This is done by calculating $L_n = F_{n-1} + F_{n+1}$. kth_Lucas handles L_1 and L_2 and calls the linear Fibonacci function, then the recursion is done by the lin_fib function. The third function is the driver code. This opens the input file, creates the output file, and initialises the Stack. The first value inside the file is stored as a variable as it is the number of iterations required for the corresponding output. A Stack was used because it is FILO, since the file is read into the Stack from the first line to the last line. This means that instead of having to reverse the order of the input after performing the required operations, the function simply pops value into the output file in the desired order. The input and output files are then closed.

Question 2:

I created three functions for Question 2 and implemented a Stack as the data structure. The first function is a Boolean check to see if the entry is an operator. The second function handles all the different operations that can be done, addition, subtraction, multiplication, and integer division. This is done by taking two inputs; the expression stack, which should only have operands, and the operation to be performed. The function pops the last two operands entered into the stack, then does the relevant operation. The last function is the driver code. This opens the input file, creates the output file, and initialises the Stack. This function uses as While loop that only breaks once it reaches the EOF. Firstly, the function takes the next entry from the RPD string, then it checks if the entry is not an operator, if it isn't, it is an operand and is pushed into the stack. Else, the entry is an operator, and the driver code calls the do operation function. This is done until the function reaches the EOF, at this point the only value left inside the stack is the final value from the RPD string, which is then written to the output file and both the input and output files are closed.

Question 3:

I created 3 functions, a Class and implemented a Queue as the data structure. Firstly, the class I made is called Position, this defines a positional grid for the knight to move on, and holds the distance travelled by the knight, the default value of distance is 0. Each time candidate knight moves, I had the current (x, y) coordinates of the knight with one of the 8 possible moves a knight can make and add one to its distance moved. The first function is a move key that converts the string notation from the input file and initialises an object Position with it's corresponding (x, y) pair. The valid_move function takes a Position and returns a Boolean, checking if the candidate move lands within the bounds of the chess board. The last function is the driver code. This opens the input file, creates the output file, and initialises the Queue. It also creates a 2D list of size 8*8 that is False for positions the knight hasn't visited and True for places it has visited. A list of the 8 Knight moves that can be made are stored as Positions with a distance of 1. The starting position is enqueued and the position is set to true on the list. The code then starts a loop that runs until a valid move sequence to the target square is found. The position at the front of the queue is dequeued, the loop then checks if the position dequeued is equal to the target square, if it is, the distance moved is written to the output file and closes the file. Otherwise, an indented for loop takes the current position of the knight and loops through the 8 moves a knight can make. It checks if each move is within the bounds of the chess board, using the valid_move function and that the square has not been visited before using the visited_squares list. If both conditions are met, the move is valid and is enqueued, and the square is set to visited in the valid_squares list. This repeats until the position dequeued equals the target square.

Question 4:

I created 2 functions and implemented a Priority Queue as the data structure. The first function I made converts the string corresponding to the importance level of the employee to an integer value for the priority. The other function is the driver code. Firstly, both the input and output files are opened, and the max operations, max submissions and max prints are stored as variables. The current number of submissions and prints are stored as variables as well. The priority queue is also initialised. A for loop is used, with the max operations value being the number of iterations done. The next request is queued into a list, where the first value in the list always corresponds to if the job is a submission or a print. If the job is to submit and the maximum number of submits has not been exceeded, a tuple of "priority, value" is enqueued into the priority queue and 1 is added to number of submissions. Else, if the job is to print and the maximum number of prints has not been exceeded, the employee with the highest priority has their print dequeued and written to the output file. This is repeated until EOF is reached, and the input and output files are closed.

Question 5:

I created 2 functions and implemented a Heap Queue as the data structure. The first function I made performs heapsort on the values and returns a sorted list of the values. The second function I made is the driver code. The input and output files are opened, and a variable final cost is initialised that will track the cost of adding any two carts together. The raw values are initially read into the heap sort function, and stored as the heap variable, which is the sorted list. In a while loop, the first two values in the heapsorted list are popped and added together. This new cart cost will be added to the final cost variable, then appended to the heap list. This list is then heapsorted again. This will run until the length of the sorted list is equal to 1, which indicates the train as had all its carts merged. At this point the final cost variable indicates the total cost of merging all the carts and it is written to the output file. The input and output files are then closed.