**Algorithms assignment**

Ryan Pitman | C23741429

Pharmaceutical manufacturing data program

Introduction to Algorithms CMPU1014

Ciaran Kelly

Table of Contents

[Project Outline 3](#_Toc162892639)

[Features: 3](#_Toc162892640)

[Task One 4](#_Toc162892641)

[Challenge 4](#_Toc162892642)

[Full Pseudocode 5](#_Toc162892643)

[Result 6](#_Toc162892644)

[Task Two 7](#_Toc162892645)

[Challenge 7](#_Toc162892646)

[Full Pseudocode 8](#_Toc162892647)

[Result 9](#_Toc162892648)

[Task Three 10](#_Toc162892649)

[Challenge 10](#_Toc162892650)

[Full Pseudocode 11](#_Toc162892651)

[Result 12](#_Toc162892652)

[Task Four 13](#_Toc162892653)

[Challenge 13](#_Toc162892654)

[Full Pseudocode 14](#_Toc162892655)

[Result 15](#_Toc162892656)

[Notable Mentions 16](#_Toc162892657)

## Project Outline

For this project, we were given a brief for a pharmaceutical manufacturing company, based in Dublin. This company has four manufacturing lines for their products, and previously was using paper-based logs for these manufacturing lines. We were tasked with creating a digital system, with an appropriate data structure, to intake these logs and grant greater functionality to the company. The requests for this program are as follows:

### Features:

* The ability to generate a report for each line, which contains information for each line of the log, sorted in the order of product id -> issue code -> date and time.
* The ability to create a report which contains all issue codes across all the companies’ manufacturing lines in a single report.
* The ability to search the inputted logs for a product id and return the earliest occurrence of that product id in the logs and the relevant issue code.
* The ability to generate a report which contains all the product ids in the logs, and the count of how many issues that product id has.

The company has provided this specification, along with a strict performance requirement for all the implementations, due to the large quantity of data they need to process. Because of this, it guides the choices of structures and algorithms we can use when implementing the specification.

## Task One

### Challenge

To implement task one, we are required to implement a report feature which shows each line log, in order of product id, issue code, and then date / time order. This feature is required to be O(NLogN) in time complexity or better. Due to this constraint, and the fact that we are required to sort this data to get the desired order, this limits us to merge sort or quick sort. I chose merge-sort for this implementation. However, to sort with the multiple criteria, I had to design a custom sorting function as opposed to merge-sorts single comparison:

**COMPARE(LINE1, LINE2)  
 IF LINE1 PRODUCT ID != LINE2 PRODUCT ID THEN  
 RETURN LINE1 PRODUCTID – LINE2 PRODUCTID**

**IF LINE1 ISSUE CODE != LINE2 ISSUE CODE THEN  
 RETURN LINE1 ISSUE CODE – LINE2 ISSUECODE**

**IF LINE1 DAY != LINE2 DAY THEN  
 RETURN LINE1 DAY – LINE2 DAY**

This logic works based on the way merge sort normally would operate. In merge sort, we check if the left number is smaller than the right number. Instead of checking that, we can check if the result of that comparison function returns a negative number, which would mean the left side is smaller based on one of the three criteria. The order of the if statements is important in this function, as they dictate the order of which the sort will happen. If the product ID is not the same, return the smaller id. If that is not the case, compare the issue code, and then the date.

### Full Pseudocode

**FOR I = 0, I < LINECOUNT THEN  
 MERGESORT(ARRAY, 0, LOGSIZE – 1)**   
  
**MERGESORT(ARRAY, LOW, HIGH)  
 IF LOW < HIGH THEN  
 MID = LOW + (HIGH – LOW) / 2  
 MERGESORT(ARRAY, LOW, MID)  
 MERGESORT(ARRAY, MID + 1, HIGH)  
 MERGE(ARRAY, LOW, MID, HIGH)  
  
MERGE(ARRAY, LOW, MID, HIGH)  
 LEFTSIZE = MID – LOW + 1  
 RIGHTSIZE = HIGH – MID**

**FOR I = 0, I < LEFTSIZE THEN  
 LEFTARR[I] = ARR[LOW + I]**

**FOR I = 0, I < RIGHTSIZE THEN  
 RIGHTARR[I] = ARR[LOW + I]**

**LEFT = 0, RIGHT = 0, INDEX = LOW**

**WHILE LEFT < LEFTSIZE AND RIGHT < RIGHTSIZE THEN  
 IF COMPARE(LEFTARR, RIGHTARR) <= 0 THEN  
 ARR[INDEX++] = LEFTARR[LEFT++]  
 ELSE ARR[INDEX++] = RIGHTARR[RIGHT++]**

**WHILE LEFT < LEFTSIZE THEN  
 ARR[INDEX++] = LEFTARR[LEFT++]**

**WHILE RIGHT < RIGHTSIZE THEN  
 ARR[INDEX++] = RIGHTARR[RIGHT++]**

### Result

The result of this operation is having four sorted arrays for each line that is inputted, all sorted based on the criteria given in the specification. Here is the output when we implement that solution in C and pass it test data:  
  
A screenshot of a computer screen

Description automatically generated

As shown in the output, the data is correctly sorted by product id, issue code and date order, and this is done in O(NLogN) time thanks to the merge-sort algorithm, which perfectly fits the specification given.

## Task Two

### Challenge

For task two of this project, the company wishes to see a general report of all their product lines, as they have started manufacturing the same products on different lines. The specification calls for a way to view issues codes for all product ids, regardless of the line they are on, and for this to be done as an O(N) operation. Due to the nature of our data, and the fact that task one sorted it correctly, we can utilize the merge portion of the merge-sort to combine each of our four line arrays into one large array:  
 A diagram of a software algorithm

Description automatically generated

### Full Pseudocode

**ISSUEREPORT(LINE ARRAY, LINE\_COUNT, LOGSIZE)  
 MERGED[TOTAL\_LOGS] = {}**

**LOG\_INDEX[LINE\_COUNT] = {0}, INDEX = 0**

**WHILE INDEX < TOTAL\_LOGS THEN  
 MIN\_INDEX = -1, MIN\_LOG = NULL**

**FOR I IN LINECOUNT THEN  
 IF LOG\_INDEX[I] < LOGSIZE THEN  
 IF COMPARE(CURRENT\_LOG, MIN\_LOG) < 0 THEN  
 MIN\_LOG = CURRENT\_LOG  
 MIN\_INDEX = I  
 IF MIN\_LOG THEN  
 MERGED[INDEX++] = MIN\_LOG  
 LOG\_INDEX[MIN\_INDEX]++  
  
 FOR I IN TOTAL LOGS THEN  
 PRINT(MERGED\_PRODUCT\_ID, MERGED\_ISSUE, MERGED\_LINE)  
  
 RETURN MERGED**

### Result

When the above logic is properly implemented in our C program, it takes input of an array which contains a sub array for each of our product lines. It iterates through this array and merges each sub array into one large array called merge. Thanks to task one, this data is perfectly sorted and as such, the merge portion of the merge sort algorithm works perfect here and has a time complexity of O(N) as per specification. However it had to be modified to merge four arrays instead of two, the time complexity stays O(N) where N is the count of logs to be merged. The resulting output looks like this: A screen shot of a code

Description automatically generated

## Task Three

### Challenge

For task three, the client wishes to have the option to search all the line logs for any given product id, and with that, the program should search through the logs for the earliest occurrence of that product id, and report back with the product id, the line it is on and the issue code for that occurrence. To solve this, we can use a custom binary search implementation, which keeps searching until the end to find the earliest occurrence, as our data is previously sorted.

**SEARCH\_ISSUE(ARR, TOTAL\_ISSUES)  
 LOW = 0, HIGH = TOTAL\_ISSUES – 1  
 GET USER\_ID FROM USER**

**WHILE LOW <= HIGH THEN  
 MID = LOW + (HIGH – LOW) / 2**

**IF ARR[MID] = USER\_ID THEN  
 HIGH = MID – 1  
 ELSE IF ARR[MID] < USER\_ID THEN  
 LOW = MID + 1  
 ELSE  
 HIGH = MID – 1**

### Full Pseudocode

**SEARCH\_ISSUE(ARR, TOTAL\_ISSUES)  
 LOW = 0, HIGH = TOTAL\_ISSUES – 1, RESULT\_INDEX = -1  
 GET USER\_ID FROM USER**

**WHILE LOW <= HIGH THEN  
 MID = LOW + (HIGH – LOW) / 2**

**IF ARR[MID] == USER\_ID THEN  
 RESULT\_INDEX = MID  
 LOW = MID – 1  
 ELSE IF ARR[MID] < USER\_ID THEN  
 LOW = MID + 1  
 ELSE  
 LOW = MID – 1  
  
 IF RESULT\_INDEX != -1 THEN  
 PRINT “EARLIEST OCCURRENCE” + ARR[RESULT\_INDEX]  
 ELSE  
 PRINT “NOT FOUND”**

### Result

The above logic, once implemented in our C program, results in the ability to search the merged lists from task 2 for the earliest occurrence of a product id. It is important that in the case that the id is found, we divide the list again on the left portion to find the earliest possible occurrence of that product id. This is the output when this logic is implemented in our program:

A close-up of a computer screen

Description automatically generated

As demonstrated in the output above, the program prompts the user for a product ID to search for and will return the earliest possible occurrence of that ID in our logs by utilizing binary search with a slight adjustment. Thanks to this binary search algorithm, this has a time complexity of O(LogN) which is what was required as per the specification given.

## Task Four

### Challenge

For the final task, the company wishes to keep a count of how many issues codes occur per a single product id, because a product can have multiple issues, this metric could give a good analysis how which areas need more focus. This can be done as a simple linear function, as our data is already sorted. Here we iterate through the data and if the product id is not unique, iterate the issue count for that product id. (This assumes that every product id will have an issue code in the log but can easily be modified if that is not the case).

**FOR I IN ISSUE\_COUNT THEN** **IF MERGED\_ARR[I] != MERGED\_ARR[I – 1] THEN** **SUMMARY\_ARR = MERGED\_ARR** **SUMMARY\_INDEX++, SUMMARY\_ISSUES++** **ELSE  
 SUMMARY\_ISSUES++**

The working logic behind this will involve creating a struct table to hold information about each product id, to keep a working total of the issue counts for that id, if the merged arrays product id is not the same as the one previous, create a new entry in the table for it and set the issue count to 0, otherwise, iterate up that issue count and move the array along one.

### Full Pseudocode

**SUMMARIZE\_PRODUCTS(MERGED, ISSUE\_COUNT)** **IF ISSUE\_COUNT = 0 THEN RETURN  
  
 SUMMARY\_INDEX = 0**

**SUMMARY SUMMARIES[TOTAL\_LOGS]**

**SUMMARIES[0].ID = MERGED.ID** **SUMMARIES[0].ISSUE\_COUNT = 1**

**FOR I IN ISSUE\_COUNT THEN** **IF MERGED[I] != MERGED[I – 1] THEN  
 SUMMARY\_INDEX++  
 SUMMARIES[SUMMARY\_INDEX].ID = MERGED.ID  
 SUMMARIES[SUMMARY\_INDEX].ISSUE\_COUNT = 1  
 ELSE  
 SUMMARIES[SUMMARY\_INDEX].ISSUE\_COUNT ++  
  
 FOR I IN SUMMARY\_INDEX THEN  
 PRINT SUMMARIES[SUMMARY\_INDEX]**

### Result

The result of this code, once implemented within our C program, is that the program will iterate through all instances of the merged data from earlier, storing product ids within a table and counting how many instances of each id show up. As mentioned previously, this assumes that all line logs will have an issue but can easily be modified to accept different criteria. The resulting function will have a time complexity of O(N), which is exactly what the specification called for in this regard. The resulting output is as follows:  
  
**A blue screen with white text

Description automatically generated**

## Notable Mentions

In this project, I was operating under a few key assumptions. Firstly, as per the brief, data will always be entered ordered by date and time, which is useful for our later functions. Secondly, I assumed that all data entered will have an issue code and an issue description in the log entry. This program can be modified to be more flexible in that area, however for simplicity’s sake and for demonstration of the algorithms at play, I left it simple.   
  
A lot of the functions within the code rely on a count of issue logs being entered, as the program needs to know how many issue lines will be entered, how many product lines there are, and so on. Some of this information is given, such as the product line count, however it is impossible to know at run time how many logs of data would be entered.  
  
A more future proof approach to this issue rather than hard coding the count of the data entries, would be to linearly count at the start of the program each entry in the line log until it hits the end, and use that for the functionality of the program. This would be a great feature if this program were to be implemented in use. Again, for the sake of simplicity, I kept the hardcoded count of how many data entries there will be in LOG\_SIZE, as I thought it not relevant to the scope of this assignment.