### Project Report for Algorithms & Design Project Jason Gaynor C23409212 **Task 1:-**

* The production line logs are either ordered by date & time or may be in a random order for each day. Prepare a report for each line in Product id, Issue code, date & time order.
* There are huge amounts of data stored, the running time of this algorithm should be O(NLog(N)) or better.

Task1:  
Design Requirements:

* Production lines ordered by Date & Time
* Prepare report for each line in product ID, Issue code and date and time order
* Running time of O(NLog(N)) or better

Task1 implementation

* Create data structure for Logs
* Generate Random data for Logs
* Use Merge Sort as it meets time complexity requirements
* Function to Display sorted logs
* Call functions in main

**Task1 Pseudocode**  
  
1. #Defines :

- MAX\_LOGS\_PER\_LINE = 10

- MAX\_LINES = 4

- MAX\_DESCRIPTION\_LENGTH = 50

2. Define QA Log struct

struct QA\_Log   
{

int line\_code

int product\_id

int issue\_code

char issue\_description[ ]

int day

int hour

int minute

}

3. Define issue\_descriptions:  
["Engine malfunction", "Wing alignment issue", "Hydraulic system failure", "Electrical component malfunction", "Fuel system problem"]

4. Define function generate\_issue\_descriptions(logs[], num\_logs):

For each log in logs:

Generate a random index between 0 and the length of issue\_descriptions - 1

String Copy the issue description at the random index into the log's issue\_description

5. Define function generate\_logs(logs[ ]):

Initialize line\_code, product\_id, issue\_code, day, hour, and minute variables

For i from 0 to MAX\_LINES \* MAX\_LOGS\_PER\_LINE - 1:

Set logs[i]'s line\_code, product\_id, issue\_code, day, hour, and minute fields

line\_code, product\_id, and issue\_code++

6. Define function merge(L[ ], R[ ], left, mid, right):

Initialize variables i, j, and k

Initialize variables n1 and n2 as the sizes of L[ ] and R[ ]

Copy data from L[ ] and R[ ] to temporary arrays

Merge the temporary arrays back into logs[left..right] based on date and time

7. Define function merge\_sort(logs[ ], left, right):

If left < right:

Calculate mid as (left + right) / 2

Recursively call merge\_sort on the left and right halves

Merge the sorted halves using the merge function

8. Define function display\_reports(logs[ ]):

For each production line from 1 to MAX\_LINES:

Print "Production Line [line number] Reports:"

For each log in logs:

If log's line\_code matches the current production line:

Print log's product\_id, issue\_code, day, hour, minute, and issue\_description

9 . In main:

Create a QA\_Log array qa\_logs[MAX\_LINES \* MAX\_LOGS\_PER\_LINE]

Call generate\_logs to generate logs

Call merge\_sort to sort logs using Merge Sort

Call display\_reports to display reports  
   
 Return 0

**Task 1 Code:**#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#define MAX\_LOGS\_PER\_LINE 10

#define MAX\_LINES 4

#define MAX\_DESCRIPTION\_LENGTH 50

// Define QA Log structure

struct QA\_Log {

    int line\_code;

    int product\_id;

    int issue\_code;

    char issue\_description[MAX\_DESCRIPTION\_LENGTH];

    int day;

    int hour;

    int minute;

};

// Define possible issue descriptions

char\* issue\_descriptions[] = {

    "Engine malfunction",

    "Wing alignment issue",

    "Hydraulic system failure",

    "Electrical component malfunction",

    "Fuel system problem"

};

// Generate random issue descriptions

void generate\_issue\_descriptions(struct QA\_Log logs[], int num\_logs) {

    for (int i = 0; i < num\_logs; i++) {

        int rand\_index = rand() % (sizeof(issue\_descriptions) / sizeof(issue\_descriptions[0]));

        strcpy(logs[i].issue\_description, issue\_descriptions[rand\_index]);

    }

}

// Generate logs for each production line

void generate\_logs(struct QA\_Log logs[]) {

    int line\_code = 1;

    int product\_id = 101;

    int issue\_code = 1;

    int day = 1;

    int hour = 0;

    int minute = 0;

    for (int i = 0; i < MAX\_LINES \* MAX\_LOGS\_PER\_LINE; i++) {

        logs[i].line\_code = line\_code++;

        logs[i].product\_id = product\_id++;

        logs[i].issue\_code = issue\_code++;

        logs[i].day = day;

        logs[i].hour = hour;

        logs[i].minute = minute;

        if (line\_code > MAX\_LINES)

            line\_code = 1;

        minute += 5; // Incrementing time by 5 minutes

        if (minute >= 60) {

            minute -= 60;

            hour++;

            if (hour >= 24) {

                hour = 0;

                day++;

            }

        }

    }

    generate\_issue\_descriptions(logs, MAX\_LINES \* MAX\_LOGS\_PER\_LINE); // Generate random issue descriptions

}

// Merge two subarrays

void merge(struct QA\_Log logs[], int left, int mid, int right) {

    int i, j, k;

    int n1 = mid - left + 1;

    int n2 = right - mid;

    // Create temporary arrays

    struct QA\_Log L[n1], R[n2];

    // Copy data to temporary arrays L[] and R[]

    for (i = 0; i < n1; i++)

        L[i] = logs[left + i];

    for (j = 0; j < n2; j++)

        R[j] = logs[mid + 1 + j];

    // Merge the temporary arrays back into logs[left..right]

    i = 0; // index of first subarray

    j = 0; // index of second subarray

    k = left; // index of merged subarray

    while (i < n1 && j < n2) {

        if (L[i].day <= R[j].day) {

            if (L[i].hour < R[j].hour || (L[i].hour == R[j].hour && L[i].minute <= R[j].minute)) {

                logs[k] = L[i];

                i++;

            } else {

                logs[k] = R[j];

                j++;

            }

        } else {

            logs[k] = R[j];

            j++;

        }

        k++;

    }

    // Copy the remaining elements of L[], if any

    while (i < n1) {

        logs[k] = L[i];

        i++;

        k++;

    }

    // Copy the remaining elements of R[], if any

    while (j < n2) {

        logs[k] = R[j];

        j++;

        k++;

    }

}

// Merge Sort function

void merge\_sort(struct QA\_Log logs[], int left, int right) {

    if (left < right) {

        int mid = left + (right - left) / 2;

        // Sort first and second halves

        merge\_sort(logs, left, mid);

        merge\_sort(logs, mid + 1, right);

        // Merge the sorted halves

        merge(logs, left, mid, right);

    }

}

// Display reports for each production line

void display\_reports(struct QA\_Log logs[]) {

    for (int line = 1; line <= MAX\_LINES; line++) {

        printf("Production Line %d Reports:\n", line);

        for (int i = 0; i < MAX\_LINES \* MAX\_LOGS\_PER\_LINE; i++) {

            if (logs[i].line\_code == line) {

                printf("Product ID: %d\n Issue Code: %d\n Date: %d\n Time: %02d:%02d\n Issue Description: %s\n",

                       logs[i].product\_id, logs[i].issue\_code, logs[i].day, logs[i].hour, logs[i].minute, logs[i].issue\_description);

            }

        }

    }

}

int main() {

    struct QA\_Log qa\_logs[MAX\_LINES \* MAX\_LOGS\_PER\_LINE];

    // Generate logs for each production line

    generate\_logs(qa\_logs);

    // Sort logs using Merge Sort

    merge\_sort(qa\_logs, 0, MAX\_LINES \* MAX\_LOGS\_PER\_LINE - 1);

    // Display reports for each production line

    display\_reports(qa\_logs);

    return 0;

}

**Task 2**

### Task 2

* Due to changes in the manufacturing process, the same product can be manufactured on different lines.
* Prepare a report which uses a *single list* to report issue codes by product Id and line Id for all production lines.
* There are huge amounts of data stored, the running time of this algorithm should be O(N) or better.

**Task 2 Design Requirements**

* Use Single List (Circular que)
* Report Issue codes by product ID and line ID for each production line
* Runtime of O(N) or better

**Task2 ideas:**  
Use the QA struct along with circular que to efficiently store the logs  
initialise the circular que - set front to 0 etc

Functions to check if queue is empty,full, enqueue, dequeue  
Generate Random logs

Display the report of issue codes by product id and line id

**Task 2 Pseudocode**

1. Define structure for QA Log

2. Define structure for Circular Queue

3. Initialize circular queue

a. Set front to 0

b. Set rear to -1

c. Set count to 0

4. Check if queue is empty

a. Return true if count is 0, false otherwise

5. Check if queue is full

a. Return true if count is equal to QUEUE\_SIZE, otherwise false

6. Enqueue an issue log into the circular queue (enqueue function)

a. Check if queue is not full

b. Update rear by incrementing it and wrapping around if necessary using modulo operation

c. Set the log at the updated rear position in the queue

d. Increment count

7. Dequeue an issue log from the circular queue (dequeue function)

a. Check if queue is not empty

b. Store the log at the front of the queue as removedLog

c. Update front by incrementing it and wrapping around if necessary using mod

d. Decrement count

e. Return removedLog

8. Main function

a. Seed random number generator

b. Initialize CircularQueue

c. Generate random logs for each production line

i. Loop for line\_code from 1 to MAX\_LINES

Loop for i from 1 to MAX\_LOGS\_PER\_LINE

1. Create a new QA\_Log log

2. Set line\_code of log to current line\_code

3. Set product\_id of log to random number between 100 and 1099

4. Set issue\_code of log to random number between 1 and 5

5. Enqueue log into issue\_queue

d. Display reports for issue codes by Product ID and Line ID

i. Print header "Issue Codes by Product ID and Line ID for all Production Lines:"

ii. Loop until issue\_queue is empty

A. Dequeue a log from issue\_queue

B. Print Product ID, Line ID, and Issue Code

Return 0

Task 2 Code:

//Jason Gaynor Algorithm’s Task2

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#define MAX\_LOGS\_PER\_LINE 10

#define MAX\_LINES 4

#define MAX\_DESCRIPTION\_LENGTH 50

#define QUEUE\_SIZE (MAX\_LINES \* MAX\_LOGS\_PER\_LINE)

// Define QA Log structure

struct QA\_Log {

    int line\_code;

    int product\_id;

    int issue\_code;

};

// Circular Queue structure for issue codes by Product ID and Line ID

struct CircularQueue {

    struct QA\_Log logs[QUEUE\_SIZE];

    int front;

    int rear;

    int count;

};

// Initialize circular queue

void init\_queue(struct CircularQueue\* queue) {

    queue->front = 0;

    queue->rear = -1;

    queue->count = 0;

}

// Check if queue is empty

int is\_empty(struct CircularQueue\* queue) {

    return queue->count == 0;

}

// Check if queue is full

int is\_full(struct CircularQueue\* queue) {

    return queue->count == QUEUE\_SIZE;

}

// Enqueue an issue log into the circular queue

void enqueue(struct CircularQueue\* queue, struct QA\_Log log) {

    if (!is\_full(queue)) {

        queue->rear = (queue->rear + 1) % QUEUE\_SIZE;

        queue->logs[queue->rear] = log;

        queue->count++;

    } else {

        printf("Queue is full. Cannot enqueue.\n");

    }

}

// Dequeue an issue log from the circular queue

struct QA\_Log dequeue(struct CircularQueue\* queue) {

    struct QA\_Log removedLog;

    if (!is\_empty(queue)) {

        removedLog = queue->logs[queue->front];

        queue->front = (queue->front + 1) % QUEUE\_SIZE;

        queue->count--;

        return removedLog;

    } else {

        printf("Queue is empty. Cannot dequeue.\n");

        removedLog.line\_code = -1; // Placeholder for empty log

        return removedLog;

    }

}

int main() {

    srand(time(NULL)); // Seed for random number generation

    struct CircularQueue issue\_queue;

    init\_queue(&issue\_queue);

    // Generate random logs for each production line

    for (int line\_code = 1; line\_code <= MAX\_LINES; line\_code++) {

        for (int i = 0; i < MAX\_LOGS\_PER\_LINE; i++) {

            struct QA\_Log log;

            log.line\_code = line\_code;

            log.product\_id = rand() % 1000 + 100; // Random product ID between 100 and 1099

            log.issue\_code = rand() % 5 + 1; // Random issue code between 1 and 5

            enqueue(&issue\_queue, log);

        }

    }

    // Display reports for issue codes by Product ID and Line ID

    printf("Issue Codes by Product ID and Line ID for all Production Lines:\n");

    while (!is\_empty(&issue\_queue)) {

        struct QA\_Log log = dequeue(&issue\_queue);

        printf("Product ID: %d, Line ID: %d, Issue Code: %d\n",

               log.product\_id, log.line\_code, log.issue\_code);

    }

    return 0;

}

**Task3**

* Provide a facility to search for the earliest occurrence of an issue code for a given product id across all production lines.
* There are huge amounts of data stored, the running time of this algorithm should be O(Log(N)) or better.

**Task3 Ideas:**

Generate Logs  
Use the QA struct again along with circl que to efficiently store the logs  
Use quick sort to sort logs  
Use Binary search to find earliest occurrence of desired issue code & Product Id  
Display earliest occurrence

**Task 3 Design Requirements**

* Search for earliest occurrence of issue Code & Product ID (Fits time complexity)
* Runtime of O(Log(N) or better  
    
    
   **Task3 pseudocode:**

1. Define struct QA\_Log

2. Define struct CircularQueu

3. Define functions: init\_queue, is\_empty, is\_full, enqueue, binary\_search\_earliest, compare\_logs.

4. Init CircularQueu

5. Generate random logs for each line and enqueue them into issue\_queue.

a. Loop through each line\_code from 1 to MAX\_LINES

i. Loop through each log from 0 to MAX\_LOGS\_PER\_LINE

Generate random log with line\_code, product\_id, and issue\_code.

Enqueue log into issue\_queue.

6. Sort logs in issue\_queue based on timestamp using compare\_logs function.

7. Perform binary search to find earliest occurrence of issue code for a given product ID.

a. Call binary\_search\_earliest with parameters: issue\_queue.logs, 0, QUEUE\_SIZE - 1, \*Product id\*, \*Issue Code\*.

b. If earliest\_index != -1:

i. Print "Earliest occurrence found at Log %d (earlest\_Index)

c. Else:

i. Print "Issue code not found for the given product ID".

Return 0  
  
**Task3 Code:**//Jason Gaynor Task3

#include <stdlib.h>

#include <time.h>

#include <stdio.h>

#define MAX\_LOGS\_PER\_LINE 10

#define MAX\_LINES 4

#define QUEUE\_SIZE (MAX\_LINES \* MAX\_LOGS\_PER\_LINE)

// Define QA Log structure with timestamp

struct QA\_Log

{

    int line\_code;

    int product\_id;

    int issue\_code;

    int timestamp; // Timestamp for binary search

};

// Circular Queue structure for issue codes by Product ID and Line ID

struct CircularQueue

{

    struct QA\_Log logs[QUEUE\_SIZE];

    int front;

    int rear;

    int count;

};

// Initialize circular queue

void init\_queue(struct CircularQueue\* queue)

{

    queue->front = 0;

    queue->rear = -1;

    queue->count = 0;

}

// Check if the Queu is empty

int is\_empty(struct CircularQueue\* queue)

{

    return queue->count == 0;

}

// Check if queue is full

int is\_full(struct CircularQueue\* queue)

{

    return queue->count == QUEUE\_SIZE;

}

// Enqueue an issue log into the circular queue with timestamp

void enqueue(struct CircularQueue\* queue, struct QA\_Log log)

{

    if (!is\_full(queue))

    {

        queue->rear = (queue->rear + 1) % QUEUE\_SIZE;

        log.timestamp = time(NULL); // Set current time as timestamp

        queue->logs[queue->rear] = log;

        queue->count++;

}   
else

    {

        printf("Queue is full. Cannot enqueue.\n");

    }

}

// Binary search for earliest occurrence of issue code for a given product ID

int binary\_search\_earliest(struct QA\_Log logs[], int left, int right, int product\_id, int issue\_code)

{

    while (left <= right)

    {

        int mid = left + (right - left) / 2;

        if (logs[mid].product\_id == product\_id && logs[mid].issue\_code == issue\_code)

        {

            while (mid > 0 && logs[mid - 1].product\_id == product\_id && logs[mid - 1].issue\_code == issue\_code)

            {

                mid--; // Move left until  earliest occurrence  found

            }

             return mid; // Return index of earliest occurrence

        }

        else if (logs[mid].product\_id < product\_id || (logs[mid].product\_id == product\_id && logs[mid].issue\_code < issue\_code))

        {

            left = mid + 1; // Search  right half

        }

        else

        {

            right = mid - 1; // Search  left half

        }

    }

    return -1; // Issue code not found for the product ID

}

// Comparison function for sorting logs based on timestamp

int compare\_logs(const void\* a, const void\* b)

{

    const struct QA\_Log\* logA = (const struct QA\_Log\*)a;

    const struct QA\_Log\* logB = (const struct QA\_Log\*)b;

    // Compare timestamps for sorting

    if (logA->timestamp < logB->timestamp) return -1;

    if (logA->timestamp > logB->timestamp) return 1;

    return 0;

}

int main()

{

    srand(time(NULL)); // Seed for random number generation

    struct CircularQueue issue\_queue;

    init\_queue(&issue\_queue);

    // Generate random logs for each production line

    for (int line\_code = 1; line\_code <= MAX\_LINES; line\_code++)

    {

        for (int i = 0; i < MAX\_LOGS\_PER\_LINE; i++)

        {

            struct QA\_Log log;

            log.line\_code = line\_code;

            log.product\_id = rand() % 100 +1; // Random product ID between 1 and 100

            log.issue\_code = rand() % 5 + 1; // Random issue code between 1 and 5

            enqueue(&issue\_queue, log);

        }

    }

    //Generate Logs to be searched

    for (int i = 0; i < QUEUE\_SIZE; i++)

    {

        printf("Log %d: Product ID %d, Issue Code %d\n", i, issue\_queue.logs[i].product\_id, issue\_queue.logs[i].issue\_code);

    }

    // Sort logs based on timestamp

    qsort(issue\_queue.logs, QUEUE\_SIZE, sizeof(struct QA\_Log), compare\_logs);

    // Binary search to find earliest occurrence of ... Eg- issue code 2 for product ID 100

    int earliest\_index = binary\_search\_earliest(issue\_queue.logs, 0, QUEUE\_SIZE - 1, 2, 100);

    if (earliest\_index != -1)

    {

        printf("Earliest occurrence found at Log: %d\n", earliest\_index);

    }

    else

    {

        printf("Issue code not found for the given product ID\n");

    }

    return 0;

}

**Runtime:**

Binary search: Runs at O(log N)

**Task4**

* Provide a report which summarises the number of issues reported for a product across all production lines.
* There are huge amounts of data stored, the running time of this algorithm should be O(N) or better.

**Task4 ideas:**

Create QA log structure again and use circular queue in order to store logs effectively.  
Generate logs and enqueue them into circular queue.  
Use Linear search to count number of issues   
Print summary report

**Task4 Pseudo:**

1. Define structure QA\_Log with fields line\_code, product\_id, issue\_code, timestamp

2. Define structure CircularQueue with array logs, front, rear, count

3. Define functions:

- init\_queue to initialize the circular queue

- is\_empty to check if the queue is empty

- is\_full to check if the queue is full

- enqueue to add logs to the queue

- count\_issues to count the number of issues reported for each product ID

4. Initialize circular que

5. Generate random logs for each line and enqueue them into issue\_queue:

a. Loop through each line\_code from 1 to MAX\_LINES

Loop through each log from 0 to MAX\_LOGS\_PER\_LINE

- Generate random log with line\_code, product\_id, and issue\_code

- Enqueue the log into issue\_queue

6. Structure to store issue counts for each product ID:

- Define structure IssueCount with fields product\_id and count

7. Count the number of issues reported for each product ID using count\_issues:

- Iterate through logs in issue\_queue

- For each log, get product\_id and update the count in issue\_counts

8. Output the summary report:

- Print Summary Report

- Loop through issue\_counts and print the product ID along with the count of issues reported for each  
  
  
  
**Task4 code:**

#include <stdlib.h>

#include <time.h>

#include <stdio.h>

#define MAX\_LOGS\_PER\_LINE 10

#define MAX\_LINES 4

#define QUEUE\_SIZE (MAX\_LINES \* MAX\_LOGS\_PER\_LINE)

// Define QA Log structure with timestamp

struct QA\_Log

{

    int line\_code;

    int product\_id;

    int issue\_code;

    int timestamp;

};

// Circular Queue structure for issue codes by Product ID and Line ID

struct CircularQueue

{

    struct QA\_Log logs[QUEUE\_SIZE];

    int front;

    int rear;

    int count;

};

// Structure to store issue counts for each product ID

struct IssueCount

{

    int product\_id;

    int count;

};

// Initialize circular queue

void init\_queue(struct CircularQueue\* queue)

{

    queue->front = 0;

    queue->rear = -1;

    queue->count = 0;

}

// Check if the queue is empty

int is\_empty(struct CircularQueue\* queue)

{

    return queue->count == 0;

}

// Check if the queue is full

int is\_full(struct CircularQueue\* queue)

{

    return queue->count == QUEUE\_SIZE;

}

// Enqueue an issue log into the circular queue with timestamp

void enqueue(struct CircularQueue\* queue, struct QA\_Log log)

{

    if (!is\_full(queue))

    {

        queue->rear = (queue->rear + 1) % QUEUE\_SIZE;

        log.timestamp = time(NULL); // Set current time as timestamp

        queue->logs[queue->rear] = log;

        queue->count++;

    } else

    {

        printf("Queue is full. Cannot enqueue.\n");

    }

}

// Linear search to count the number of issues reported for each product ID

void count\_issues(struct CircularQueue\* queue, struct IssueCount\* issue\_counts, int\* num\_products)

{

    for (int i = 0; i < queue->count; i++) {

        int product\_id = queue->logs[i].product\_id;

        int found = 0;

        for (int j = 0; j < \*num\_products; j++)

        {

            if (issue\_counts[j].product\_id == product\_id)

            {

                issue\_counts[j].count++;

                found = 1;

                break;

            }

        }

        if (!found)

        {

            issue\_counts[\*num\_products].product\_id = product\_id;

            issue\_counts[\*num\_products].count = 1;

            (\*num\_products)++;

        }

    }

}

int main()

{

    srand(time(NULL)); // Seed for random number generation

    struct CircularQueue issue\_queue;

    init\_queue(&issue\_queue);

    // Generate random logs for each production line

    for (int line\_code = 1; line\_code <= MAX\_LINES; line\_code++)

    {

        for (int i = 0; i < MAX\_LOGS\_PER\_LINE; i++)

        {

            struct QA\_Log log;

            log.line\_code = line\_code;

            log.product\_id = rand() % 100 + 1; // Random product ID between 1 and 100

            log.issue\_code = rand() % 5 + 1;  // Random issue code between 1 and 5

            enqueue(&issue\_queue, log);

        }

    }

    // Generate Logs to be searched

    for (int i = 0; i < QUEUE\_SIZE; i++)

    {

        printf("Log %d: Product ID %d, Issue Code %d\n", i, issue\_queue.logs[i].product\_id, issue\_queue.logs[i].issue\_code);

    }

    // Structure to store issue counts for each product ID

    struct IssueCount issue\_counts[QUEUE\_SIZE];

    int num\_products = 0;

    // Count the number of issues reported for each product ID

    count\_issues(&issue\_queue, issue\_counts, &num\_products);

    // Output the summary report

    printf("Summary Report:\n");

    for (int i = 0; i < num\_products; i++)

    {

        printf("Product ID %d: %d issues reported\n", issue\_counts[i].product\_id, issue\_counts[i].count);

    }

    return 0;

}

**Time complexity:**  
Meets time Complexity of O(N) As I used Linear search which has a time Complexity of O(N)