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:

```c
#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>
```

```c
#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");
    }
}
```

```c
// 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:

```c
//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;
}
```

```c
// 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
}
```

```c
// 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)