Utilising Data Structures for Effective Operating Systems

Designing Solutions to Various Algorithmic Issues

Module: Data Structures and Algorithms

Focus: Solving Problems Using Algorithms

A report originally created to fulfil the degree of

BSc (Hons) in Computer Science

University of Brighton

Submitted 07/01/2019

Abstract

This report assesses how the choice of data structures and algorithms impact the performance of programs. An appropriate data structure has been designed in order to create a program utilising queues and linked lists, through the use of object-oriented design principles when required.

Contents

Task 1: Designing an Eight Queens Solution

The first action I took when completing this task was creating the guidelines that I needed to follow in order to adhere to the specification. This included having an input for the user to choose the first queen location, a good physical representation of each queen's location and the ability to remove previous queens if needed by backtracking. I specifically choose to use the backtracking algorithm as I wanted my code to be as fast and efficient as possible, and after developing various ways of making this happen, I believed this to be the best way to achieve this goal. In order to successfully deduce which was the next available space to place a queen, I created three rules for my program to follow:

- Queens cannot be located in the same column.
- Queens cannot be located in the same row.
- Queens cannot be located in the same diagonal.

I decided to store the column and row values of each queen, so that I could simply iterate through each column until possible locations were found. If at any point the location being searched is discovered to be invalid, the program breaks out of that section of code and skips straight to searching the next space. If the program reaches a point where no more queens can be placed, the previous queen is removed. The program will then continue search the next available space, until all eight queens can be placed. By keeping a record of the previous queens and only backtracking by one instead of starting from scratch, the time it takes to produce a solution is significantly reduced.

To help myself and anyone else reading my code, I added comments for all the major function in the program. This allows me to showcase what each step of the program does, making it easier for me to see which sections of code I needed to focus on when trying to debug my program. One main issue I had when designing my code when approaching this task was looping the search back to the start of the board when the program had reached the final column. I originally wrote the program to only search until the final column, however I did not account for the fact users may decide not to place the queen in the first column; detecting queens to the left diagonally took some extra thinking.

I decided the best way to target the problem was to split the original third rule into four separate rules, by search the upper left, lower left, lower right, and upper right diagonals instead. One simple way I could utilise this in the future to further the efficiency of my program could be to detect which corner the current location is closed to and starting with the corresponding diagonal. For example, if a location near the top left-hand corner is being searched, there is a more likely chance of a queen being in the bottom right diagonal due to the increased number of spaces compared to the top left.

There are a few additional design choices I could add if I were to attempt this task again in the future. For example, to allow for every possibility once the user has inputted the first queen's location, you could continue to backtrack until no more possible locations are left. This would be done by printing out each solution once the program has placed all eight queens, but then continue to backtrack the final queen each time a solution is found. Once every solution is discovered, the program would then terminate after printing out how many solutions had been found.

Another change I could make to the program could be to all the user to choose how many queens the user would want to place or make the size of the board a variable that the user could change. Ultimately, I decided to focus all my efforts on following the guidelines set out for me in order to achieve a comprehensive solution in the allotted time. Overall, I am very satisfied with the result, due to the efficiency based on the algorithm I chose to use and the overall design of the program.

Main Class

This program is designed to solve the eight queen's problem using backtracking. The program begins with the user inputting the column and row of the first queen. The program then calculates all of the resulting board positions for the remaining queens. If it reaches the end without finding all 8 queens, the previous queen is removed. A new location is then considered, and the program moves forward again. For each queen, the side is checked, followed by an upper left diagonal check; The remaining queens are then completed in an anti-clockwise fashion until all 8 queens have been placed.

```
package dataPackage;

import java.util.Scanner;

public class Main {

    // Prints a visual representation of each queen location to the console.

    void printSolution(int board[][]) {

        System.out.println("\n/ A B C D E F G H");

        for (int i = 0; i < 8; i++) {
            System.out.print(8 - i + " ");

            for (int j = 0; j < 8; j++) {
                System.out.print(board[i][j] + " ");
            }

            System.out.println();
        }
    }

    // Checks for the next possible queen location, based on the previous ones placed.

    boolean validLocation(int board[][], int row, int column) {

        int i, j, counter;

        // Checks to see if a queen is located on the same row

        for (i = column, counter = 0; counter < 8; i++, counter++) {
            if (i >= 8) {
                i -= 8;
            }

            if (board[row][i] == 1) {
                System.out.println("Column = " + (column+1) + ", Row = " + (row+1) + "
failed, already a queen to the side.");
                return false;
            }
        }

        // Checks to see if a queen is located on the upper left diagonal.
```

```
for (i = row, j = column; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 1) {
                System.out.println("Column = " + (column+1) + ", Row = " + (row+1) + "
failed, already a queen in the upper left diagonal.");
                return false;
            }
        }

        // Checks to see if a queen is located on the lower left diagonal

        for (i = row, j = column; j < 8 && i >= 0; i--, j++) {
            if (board[i][j] == 1) {
                System.out.println("Column = " + (column+1) + ", Row = " + (row+1) + "
failed, already a queen in the lower left diagonal.");
                return false;
            }
        }

        // Checks to see if a queen is located on the lower right diagonal

        for (i = row, j = column; i < 8 && j < 8; i++, j++) {
            if (board[i][j] == 1) {
                System.out.println("Column = " + (column+1) + ", Row = " + (row+1) + "
failed, already a queen in the lower right diagonal.");
                return false;
            }
        }

        // Checks to see if a queen is located on the upper right diagonal

        for (i = row, j = column; j >= 0 && i < 8; i++, j--) {
            if (board[i][j] == 1) {
                System.out.println("Column = " + (column+1) + ", Row = " + (row+1) + "
failed, already a queen in the upper right diagonal.");
                return false;
            }
        }

        System.out.println("Column = " + (column + 1) + ", Row = " + (row + 1) + "
placed.");
        return true;
    }

    boolean backtrackFunction(int board[][], int column, int userQueenRow, int placed) {

        // This stops the continuous loop if all 8 queens have been placed on the board.

        if (placed >= 8) {
            System.out.println("Total Queens Placed: " + placed);
            return true;
        }

        int y = column;

        if (column >= 8) {
            y = column - 8;
        }

        System.out.println("Total Queens Placed: " + placed);

        // Tries placing a queen in this column by searching all the rows for locations.
        for (int i = userQueenRow; i < userQueenRow + 8; i++) {

            // If i goes over 7, this allows the program to loop back to the beginning.

            int x = i;

            if (i >= 8) {
                x = i - 8;
            }

            int xPrinted = x + 1;
            int yPrinted = y + 1;

            // If a location is valid, this places a queen down in that spot.

            if (validLocation(board, x, y)) {
                board[x][y] = 1;
                placed += 1;

            // This allows the rest of the queens to be placed.

            if (backtrackFunction(board, yPrinted, xPrinted, placed) == true) {
                return true;
            }

            // If it reaches the end without all queens, backtrack to remove the last one.

            System.out.println("Column = " + (yPrinted) + ", Row = " + (xPrinted) + "
removed.");
            board[x][y] = 0;
            placed -= 1;
            }
        }
        return false;
    }

    boolean solveNQ(int userQueenColumn, int userQueenRow, int placed) {

        int board[][] = {{0, 0, 0, 0, 0, 0, 0, 0},
                         {0, 0, 0, 0, 0, 0, 0, 0},
                         {0, 0, 0, 0, 0, 0, 0, 0},
                         {0, 0, 0, 0, 0, 0, 0, 0},
                         {0, 0, 0, 0, 0, 0, 0, 0},
                         {0, 0, 0, 0, 0, 0, 0, 0},
                         {0, 0, 0, 0, 0, 0, 0, 0},
                         {0, 0, 0, 0, 0, 0, 0, 0}};

        if (backtrackFunction(board, userQueenColumn, userQueenRow, placed) == false) {
            System.out.print("Error, no solution detected.");
            return false;
        }

        printSolution(board);
        return true;
    }
    public static void main(String args[]) {

        // This allows the user to input the first location of the queen.

        Scanner reader = new Scanner(System.in);
        System.out.print("Enter the column number of your queen: ");
        int userQueenColumn = reader.nextInt() - 1;
        System.out.print("Enter the row number of your queen: ");
        int userQueenRow = reader.nextInt() - 1;
        reader.close();

        int placed = 0;

        // This starts a timer to see how long it takes to produce a viable result.

        long tStart = System.currentTimeMillis();

        // Runs the program to produce a solution based on the first queen location.

        Main NQueensProgram = new Main();
        NQueensProgram.solveNQ(userQueenColumn, userQueenRow, placed);

        // This ends the previously mentioned timer.

        long tEnd = System.currentTimeMillis();
        long tDelta = tEnd - tStart;

        // This works out how long it has taken to produce a viable result.

        System.out.println("\nTime taken to produce a solution: " + tDelta + "
milliseconds");
    }
}
```

Task 2: Designing an IT Ticketing System

For this task I have decided to use linked lists, a dynamic data structure that can grow and shrink during a program's runtime, which is done by allocating and deallocating memory. There is no need to give an initial size to the linked list, so there is no memory wastage as the list size can change as memory is only allocated when it is required. One potential downside to this is that more memory is required to store the elements themselves due to each node containing a pointer. If I were to use an array, however, then there could be a lot of memory wastage if the array is too large for the number of elements stored instead it. Since an IT ticketing system will not have a set number of tickets at any given time, the advantage from the linked list is greater than the downside of using more memory for the pointers, as less memory is usually required overall, provided the code is written efficiently.

Inserting and deleting nodes using linked lists was very easy to do, as unlike an array I did not need to shift the elements after the insertion or deletion of an element. All I needed to do with the linked list was update the address that is present in the next pointer of a node. This makes developing data structures such as stack and queues easily to implement using linked list. Traversing back through a linked list can be difficult to implement, however, without the use of a doubly linked list. If I were to implement a doubly linked list, then a great deal of extra memory would be required, resulting in a waste of memory. While a doubly linked list might have been more useful for other applications, the benefits did not outweigh the extra memory required, so I decided to stick with a singly linked list.

Main Class

This class simulates a ticket queueing system that sorts out submitted tickets by priority. Each ticket contains a unique ID, a description of the issue, a creator, an owner (the person handling the ticket), and a priority. This ranges from 1 (most important) to 4 (least important), with higher priority tickets being dealt with before less important issues. Once a ticket is resolved, it is then removed from the ticket system, where the next more important ticket is pushed to the top.

```
package dataPackage;

public class Main {

    public static void main(String[] args) {

    PriorityQueue ticketQueue = new PriorityQueue();

    Ticket ticketJay = new Ticket(1, "I think I have a virus, my PC's so slow since
downloading a file.", "Jay Massey", "Adam Tyler", 1);
    Ticket ticketBill = new Ticket(2, "A network issue is not letting me upload any of my
work.", "Bill Woods", "Reece Tennant", 2);
    Ticket ticketHenry = new Ticket(3, "This software isn't working since the new update.",
"Henry Adams", "Adam Tyler", 3);
    Ticket ticketSteve = new Ticket(4, "My new computer has arrived and needs setting up.",
"Steve Banks", "Reece Tennant", 4);
    Ticket ticketJoe = new Ticket(5, "The new maintenance guy doesn't know what he's doing,
can you help?", "Joe Mayes", "Adam Tyler", 4);
    Ticket ticketDave = new Ticket(6, "My new mouse isn't working, can't I just get a new
one?", "Dave Turner", "Reece Tennant", 4);
    Ticket ticketAmy = new Ticket(7, "I think someone is logging onto my computer
remotely.", "Amy Hills", "Reece Tennant", 1);
```

```
    // Inserting example tickets into the system.

    ticketQueue.insert(ticketJay);     //
    ticketQueue.insert(ticketBill);    // These tickets are inserted into the ticket queue,
    ticketQueue.insert(ticketHenry);   // where they are then into the ticket queue and are
    ticketQueue.insert(ticketSteve);   // sorted based on their priority levels, with the
    ticketQueue.insert(ticketJoe);     // higher priority tickets placed at the top.
    ticketQueue.insert(ticketDave);    //

    // Displaying all the tickets in the system.

    ticketQueue.displayAll();          // Displays the unsolved tickets currently queued.

    // Displaying and removing the most important ticket in the queue.

    ticketQueue.displayTop();          // Displays the top ticket in the queue.
    ticketQueue.removeTop();           // Removes the top ticket from the queue.

    // Removing a specific ticket in the queue, using the unique ID that corresponds to it.

    ticketQueue.removeTicket(4);       // Remove the ticket with the corresponding ID.

    // Search for specific ticket in the queue, using the unique ID that corresponds to it.

    ticketQueue.searchTicket(3);     // Searches for the ticket with the corresponding ID.

    // Displaying and removing the most important ticket in the queue.

    ticketQueue.displayTop();
    ticketQueue.removeTop();

    // Changing the priority of an already existing ticket.

    ticketQueue.displayTop();
    ticketQueue.displayAll();
    ticketQueue.changePriority(5, "That new guy was awful, he's made my PC situation worse
by somehow disabling my network.", 2);
    ticketQueue.displayTop();
    ticketQueue.displayAll();

    // Inserting a new ticket into the system, where it is placed based on its priority.

    ticketQueue.insert(ticketAmy);
    ticketQueue.displayTop();
    ticketQueue.displayAll();

    }

}
```

Priority Class

This class merges the 4 individual queues to create a priority queue.

```
package dataPackage;

public class PriorityQueue {

    private Queue[] queues;  // This makes the variable accessible only inside this class.

    public PriorityQueue() {

        queues = new Queue[4];              // Creates 4 queues with the class Queue.
        queues[0] = new Queue();            // Creates a queue for priority 1 tickets.
        queues[1] = new Queue();            // Creates a queue for priority 2 tickets.
        queues[2] = new Queue();            // Creates a queue for priority 3 tickets.
        queues[3] = new Queue();            // Creates a queue for priority 4 tickets.
    }

    public void changePriority(int id, String newDesc, int newPrior) {

        for(int q = 0; q < 4; q++) {

            if(queues[q].ticketPresent()) {        // If the current queue is not empty.

                Ticket change = queues[q].changePriority(id, newDesc, newPrior);
                if (change != null) {

                    Ticket newTicket = new Ticket(change.getID(),
                    change.getDescription(), change.getCreator(), change.getHandler(),
                    change.getPriority());  // This is done so only the first ticket in
                                            // "change" is inserted back in the queue.
                    this.insert(newTicket);
                    break;
                }
            }
        }
    }

    public void displayAll() {

        System.out.println("\nAll Remaining Tickets: (" + queueLength() + ")\n");
        for(int q = 0; q < 4; q++) {
            if(queues[q].ticketPresent()) {
                queues[q].displayAll();     // This displays all the tickets in the queue.
            }
        }
    }

    public void displayTop() {

        System.out.println("\nNext Ticket:");
        for(int q = 0; q < 4; q++) {
            if(queues[q].ticketPresent()) {
                queues[q].displayTop();            // Causes top queued ticket to display.
                break;                             // Stops more tickets being printed.
            }
        }
    }
    public void insert(Ticket ticket) {

        switch(ticket.getPriority()) {              // Finds priority to determine queue.
            case 1:
                queues[0].insert(ticket);           // Inserts a new priority 1 ticket.
                break;
            case 2:
                queues[1].insert(ticket);           // Inserts a new priority 2 ticket.
                break;
            case 3:
                queues[2].insert(ticket);           // Inserts a new priority 3 ticket.
                break;
            case 4:
                queues[3].insert(ticket);           // Inserts a new priority 4 ticket.
                break;
        }
    }

    public int queueLength() {              // Finds the number of tickets in ticket queue.

        int length = 0;
        for(int q = 0; q < 4; q++) {
            if(queues[q].ticketPresent()) {
                length += queues[q].ticketCycle();         // Adds 1 for each queued ticket.
            }
        }

        return length;       // Returns the total number of tickets in the entire system.
    }

    public void removeTicket(int removeID) {

        for(int q = 0; q < 4; q++) {
            if(queues[q].ticketPresent()) {                // The 0 in this function tells
                queues[q].removeTicket(removeID, 0);       // the system to print that the
            }                                              // ticket has been removed.
        }
    }

    public void removeTop() {

        for(int q = 0; q < 4; q++) {
            if(queues[q].ticketPresent()) {
                queues[q].removeTop();
                break;
            }
        }
    }

    public void searchTicket(int searchID) {

        System.out.println("\nSearching for Ticket " + searchID + ":");
        for(int q = 0; q < 4; q++) {
            if(queues[q].ticketPresent()) {
                queues[q].searchTicket(searchID);  // Checks for the user requested ticket.
            }
        }
    }
}
```

Queue Class

This class constructs a queueing system, adding values to the end and removes items from the front.

```
package dataPackage;


public class Queue {

    private Ticket head;
    private Ticket tail;

    public Queue() {

        head = null;
        tail = null;
    }

    public Ticket changePriority(int inputedID, String newDesc, int newPrior) {

        Ticket current = head;
        while (current != null) {           // Loops until the end of the list.

            if (inputedID == current.getID()) {

                current.changePriority(newDesc, newPrior);       // Updates the ticket.
                System.out.println("\nTicket " + inputedID + " has been updated to 
                priority " + newPrior + ".");
                removeTicket(current.getID(), 1);
                return current;

            } else {

                current = current.getNext();       // Updates ticket to next in the list.
            }
        }

        return null;
    }

    public void displayAll() {

        Ticket current = head;
        while (current != null) {           // Loops until the end of the list.

            current.displayTicket();        // Displays current ticket in the list.
            current = current.getNext();
        }
    }

    public void displayTop() {

        head.displayTicket();
    }
```

```java
public void insert(Ticket ticket) {

    if(head == null && tail == null) {          // If queue is empty, execute this code.

        head = ticket;                          // Sets the head to the inputted ticket.
        tail = ticket;                          // Sets the tail to the inputted ticket.

    } else {

        tail.setNext(ticket);                   // Sets the next ticket in the tail.
        tail = ticket;                          // Sets the tail to the inputted ticket.
    }
}

public boolean isEmpty() {

    return head == null;                        // Checks if the ticket queue is empty.
}

public Ticket removeTicket(int removeID, int removeType) {

    Ticket current = head;
    if (current.getID() == removeID) {

        if (current.getNext() == null) {

            head = null;
            tail = null;
            return current;

        } else {
                                                // If true, tell the system to print
            if (removeType == 0) {              // that the ticket has been removed.

                System.out.println("\nTicket " + current.getID() + " removed.");
            }

            head = current.getNext();
            return current;
        }

    }

    while (current.getNext() != null) {         // Loops until the end of the list.

        int checkID = current.getNext().getID();
        if (checkID == removeID) {

            current.setNext(current.getNext().getNext());
            System.out.println("\nTicket " + checkID + " removed.");
            return current;

        } else {

            current = current.getNext();
        }
    }

    return current;
}
```

```
    public Ticket removeTop() {

        Ticket ticket = head;
        System.out.println("\nTicket " + ticket.getID() + " completed." );
        if(head == tail) {          // If one ticket only, set both head and tail to null.

            head = null;
            tail = null;

        } else {

            head = head.getNext();
        }

        return ticket;
    }

    public Ticket searchTicket(int inputedID) {

        Ticket current = head;
        while (current != null) {          // Loops until the end of the list.

            if (inputedID == current.getID()) {

                current.displayTicket();
                return current;

            } else {

                current = current.getNext();
            }
        }

        return null;
    }

    public int ticketCycle() {

        int cycleLength = 0;
        Ticket current = head;
        while (current != null) {          // Loops until the end of the list.

            current = current.getNext();
            cycleLength += 1;
        }

        return cycleLength;                // Returns how many tickets are in a queue.
    }

    public boolean ticketPresent() {

        return head != null;               // Checks if there is a ticket in the queue.
    }
}
```

Ticket Class

This class is used to fetch data relating to each ticket processed in the system.

```java
package dataPackage;

public class Ticket {

    private int ID;                 // A unique ID for the ticket
    private String description;     // A description of the problem
    private String creator;         // The creator of the problem
    private String handler;         // The handler of the problem
    private int priority;           // The priority of the ticket
    private Ticket next;            // The next ticket pointer

    public Ticket(int id, String desc, String create, String handle, int rank) {
        ID = id;
        description = desc;
        creator = create;
        handler = handle;
        priority = rank;
    }

    public void changePriority(String newDesc, int newPrior) {  // Changes ticket priority.
        description = newDesc; priority = newPrior;
    }

    public void displayTicket() {   // This is the displayed structure for each ticket.
        System.out.printf("Ticket ID: " + ID + ", Description: '" + description + "',
        Creator: " + creator + ", Handler: " + handler + ", Priority: " + priority + "\n");
    }

    public String getCreator() {
        return creator;      // Returns the name of the person submitting the ticket.
    }

    public String getDescription() {
        return description;  // Returns the ticket description.
    }

    public String getHandler() {
        return handler;      // Returns the name of the person dealing with the ticket.
    }

    public int getID() {
        return ID;           // Returns the ticket ID.
    }

    public Ticket getNext() {
        return next;         // Returns the next ticket pointer.
    }

    public int getPriority() {
        return priority;     // Returns the priority of the ticket.
    }

    public void setNext(Ticket ticket) {
        next = ticket;       // Sets the next ticket pointer.
    }
}
```

Task 3: Designing a Process Scheduler

In order to create an effective process scheduler, I initially decided to work out the most efficient algorithm for the task. I considered various scheduling algorithms, such as First Come First Serve (FCFS), Shortest Job First (SJF) and Priority Scheduling. While each of these algorithms worked well in certain scenarios, I felt like they did not achieve the efficiency I was looking for. Since the system requirements state that there are three priority levels, I decided not to use FCFS and SJF, as I felt they ran the risk of long processes not being completed for a vast amount of time.

I initially ended up choosing to use Priority Scheduling, but I knew I would still have a low priority starvation problem if I just used a basic version of the algorithm. To combat this, I created an aging system that adds points to all processes that have not yet been completed. When a process initially arrives, its age is tracked until it has finished being processed. When a process has an age greater than a set limit, the amount of points attributed to it means that it will have reached the top of the priority queue. The number for the limit is calculated by multiplying the average burst time by the number of scheduled processes that have been completed. This means that the more processes arrive in the system, the more data the system has to working out what the average burst time is.

Since I have assumed that only an estimate of the burst time is known (to simulate the fact that in reality the exact number would be unknown), this is also extremely helpful in working out the order of processes when longer processes enter the system. If a process is taking longer than average burst time, then the system knows that it must be close to competition and therefore continues to process it for the final few bursts, rather than switching to a new process. This prevents long process starvation, as longer processes are allowed to finish being processed when previously they were not.

These solutions decrease the average waiting and turnaround time, resulting in more processes being completed quicker than they were before. To showcase this, I also produced a spreadsheet that showcases the various statistics of the algorithm further, which demonstrates how it changes and adapts to achieve the most efficient output based on the data that is inputted. If I were to attempt this task again in the future, I would change the number of processes to be dynamic rather than only allowing a maximum of 5 processes in the Java project, as the algorithm itself is already dynamic. Overall, I am happy with the various scheduling algorithms that I have learned, and how I have utilised them in order to develop a process scheduler that is effective and efficient.

Excel Spreadsheet

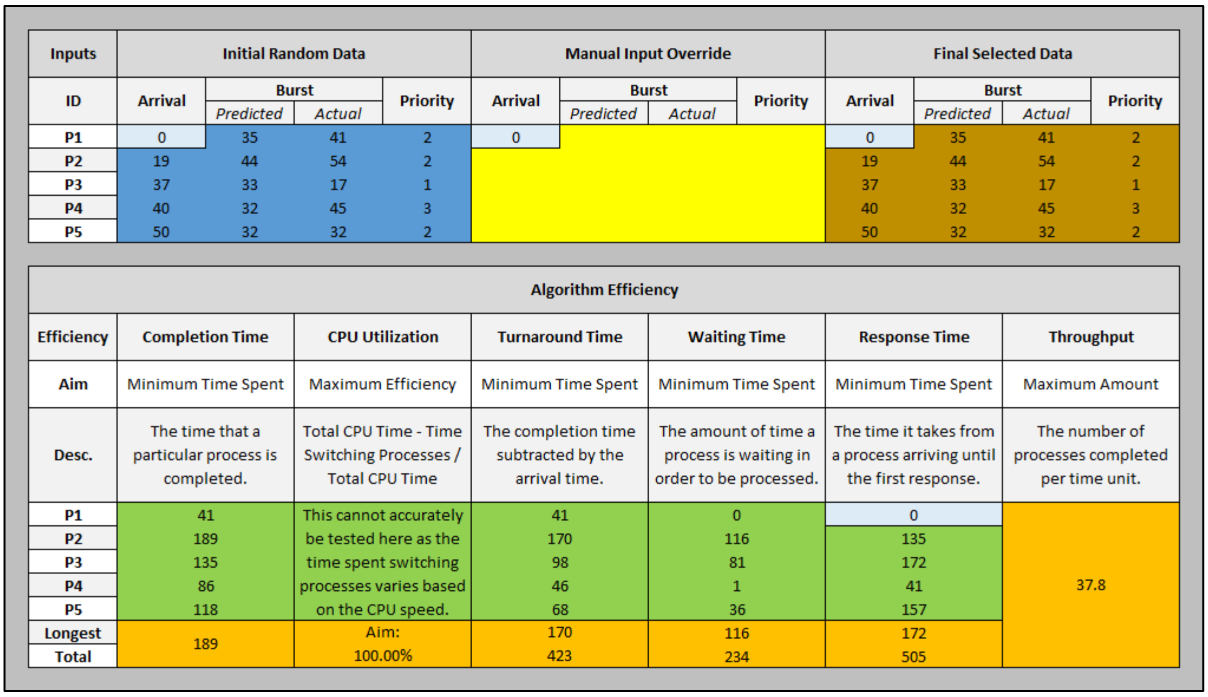
To showcase multiple scenarios that can occur whilst using this algorithm, this spreadsheet is set to generate random inputs. The arrival times for each subsequent process after P1 are all chosen by selecting a random number between 1 and 20 and adding it to the arrival time of the previous process. A random number between 1 and 50 is then chosen for the predicted burst time of each process, and a random value between -50% and 50% of this prediction is added to form an actual burst time. This is to simulate the fact that it would be impossible to calculate the exact burst time in reality. The last number generated is the process priority, which is random number between 1 and 3. To showcase the efficiency of the algorithm that I developed, I utilised the following values:

- Completion Time: The time that a particular process is completed.
- Turnaround Time: The completion time subtracted by the arrival time.
- Waiting Time: The amount of time a process is waiting in order to be processed.
- Response Time: The time it takes from a process arriving until the first response.

These are the factors I kept in mind when deciding how I should go about making the most efficient process scheduler. To do this, each process is executed based on a calculated score, which considers its size and age in comparison to the previous processes. The spreadsheet showcased below that outlines this design and breaks down the process of each step is colour coded as followed:

- Dark Blue: Initial Random Data.
- Yellow: Manual Input Override (Can be left blank).
- Brown: Final Selected Data (Manual inputs if present, otherwise random data is used).
- Light Blue: Cells that will always equal 0.
- Green: Calculation in Progress.
- Orange: Calculation Completed.

Algorithm Efficiency

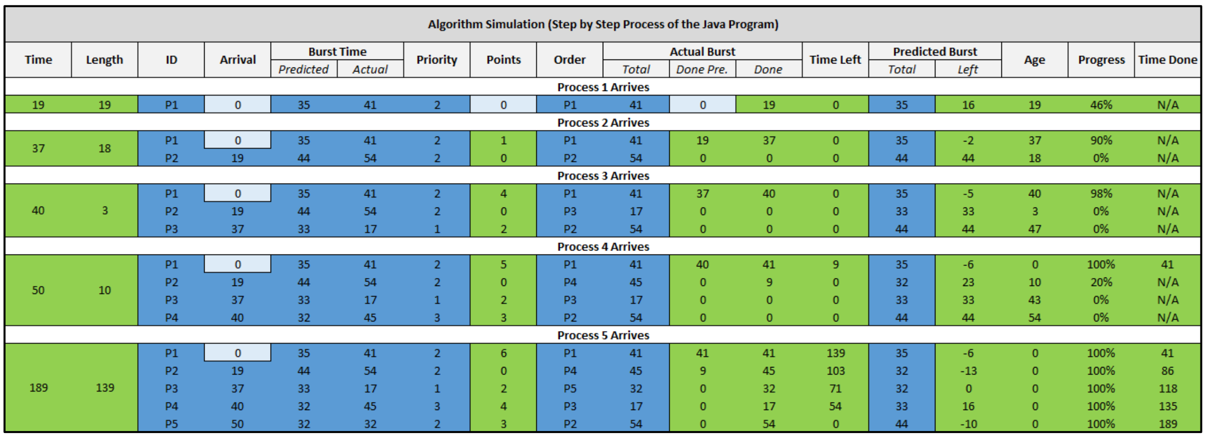


Inputs	Initial Random Data				Manual Input Override				Final Selected Data			
ID	Arrival	Burst		Priority	Arrival	Burst		Priority	Arrival	Burst		Priority
		Predicted	*Actual*			*Predicted*	*Actual*			*Predicted*	*Actual*	
P1	0	35	41	2	0				0	35	41	2
P2	19	44	54	2					19	44	54	2
P3	37	33	17	1					37	33	17	1
P4	40	32	45	3					40	32	45	3
P5	50	32	32	2					50	32	32	2

Algorithm Efficiency						
Efficiency	Completion Time	CPU Utilization	Turnaround Time	Waiting Time	Response Time	Throughput
Aim	Minimum Time Spent	Maximum Efficiency	Minimum Time Spent	Minimum Time Spent	Minimum Time Spent	Maximum Amount
Desc.	The time that a particular process is completed.	Total CPU Time - Time Switching Processes / Total CPU Time	The completion time subtracted by the arrival time.	The amount of time a process is waiting in order to be processed.	The time it takes from a process arriving until the first response.	The number of processes completed per time unit.
P1	41	This cannot accurately be tested here as the time spent switching processes varies based on the CPU speed.	41	0	0	37.8
P2	189		170	116	135	
P3	135		98	81	172	
P4	86		46	1	41	
P5	118		68	36	157	
Longest	189	Aim:	170	116	172	
Total		100.00%	423	234	505	

Figure 1: *An example of the algorithm using random inputs, breaking it down into useable variables.*

Algorithm Simulation



Algorithm Simulation (Step by Step Process of the Java Program)

Time	Length	ID	Arrival	Burst Time		Priority	Points	Order	Actual Burst			Time Left	Predicted Burst		Age	Progress	Time Done
				Predicted	*Actual*				*Total*	*Done Pre.*	*Done*		*Total*	*Left*			
Process 1 Arrives																	
19	19	P1	0	35	41	2	0	P1	41	0	19	0	35	16	19	46%	N/A
Process 2 Arrives																	
37	18	P1	0	35	41	2	1	P1	41	19	37	0	35	-2	37	90%	N/A
		P2	19	44	54	2	0	P2	54	0	0	0	44	44	18	0%	N/A
Process 3 Arrives																	
40	3	P1	0	35	41	2	4	P1	41	37	40	0	35	-5	40	98%	N/A
		P2	19	44	54	2	0	P3	17	0	0	0	33	33	3	0%	N/A
		P3	37	33	17	1	2	P2	54	0	0	0	44	44	47	0%	N/A
Process 4 Arrives																	
50	10	P1	0	35	41	2	5	P1	41	40	41	9	35	-6	0	100%	41
		P2	19	44	54	2	0	P4	45	0	9	0	32	23	10	20%	N/A
		P3	37	33	17	1	2	P3	17	0	0	0	33	33	43	0%	N/A
		P4	40	32	45	3	3	P2	54	0	0	0	44	44	54	0%	N/A
Process 5 Arrives																	
189	139	P1	0	35	41	2	6	P1	41	41	41	139	35	-6	0	100%	41
		P2	19	44	54	2	0	P4	45	9	45	103	32	-13	0	100%	86
		P3	37	33	17	1	2	P5	32	0	32	71	32	0	0	100%	118
		P4	40	32	45	3	4	P3	17	0	17	54	33	16	0	100%	135
		P5	50	32	32	2	3	P2	54	0	54	0	44	-10	0	100%	189

Figure 2: *The resulting step by step process of the process scheduler, using the data from Figure 1.*

Description of Algorithm Variables		
Time		The time *(in ms)* that the next process arrives. This is when the algorithm determines if the process should continue based on the factors below.
Length		This is how long *(in ms)* until the next process arrives (or if all processes have arrived, until they have all been completed).
Priority		A priority level of 1 is high, 2 is medium and 3 is low.
Points		The higher the priority, the more points allocated. Processes that have been around longer than the average burst time are given more points.
Order		The process with the highest amount of points is calculated to be the most efficient process to work on next.
Actual Burst	*Total*	The time it will take to complete a process *(in ms, this number is only used if it is known, otherwise it is compared to the average burst time).*
	Done Pre.	The amount of time *(in ms)* spent on the process so far.
	Done	The amount of time *(in ms)* spent on the process at the time the next process arrives.
Time Left		The time left to complete a process *(in ms, not used in any calculations as it would be unknown, this is simply to represent the algorithm above).*
Predicted Burst	*Total*	The amount of time *(in ms)* that the process is predicted to take.
	Left	The amount of time *(in ms)* left before the process is completed.
Age		This is how much time *(in ms)* the process has spent waiting to be completed. Finished processes have their age set back to 0.
Progress		The current status of each process when the next one arrives *(this would also be unknown and is just used to showcase the algorithm efficiency).*
Time Done		The time *(in ms)* that the current process finished. If it is still ongoing, a value of "N/A" is displayed instead.

Main Class

```
package dataPackage;

import java.io.BufferedReader;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.ArrayList;

public class Main {

    public static void displayTitles() {

        System.out.print("\n| Process ID | Arrival Time | Predicted Burst Time | Priority |
Score | Amount Worked |\n"
                        + "|-------------------------------------------------------------
-----------------------|\n");
    }

    public static void main(String args[])
    {
        boolean algorithmRunning = true;
        int currentLoopIteration = 0;
        int numOfProcessesArrived = 0;

        ArrayList<Process> processes = readProcessesFromCSV("src/dataPackage/data.csv");

        Process process1 = processes.get(0);
        Process process2 = processes.get(1);
        Process process3 = processes.get(2);
        Process process4 = processes.get(3);
        Process process5 = processes.get(4);

        Process position1 = process1;
        Process position2 = process2;
        Process position3 = process3;
        Process position4 = process4;
        Process position5 = process5;

        ArrayList<Process> positions = new ArrayList<Process>();

        positions.add(position1);
        positions.add(position2);
        positions.add(position3);
        positions.add(position4);
        positions.add(position5);

        int totalProcesses = 5;

        while (algorithmRunning == true) {

            int processesCompleted = 0;
            int totalBurstsCompleted = 0;

            for (int i = 0; i < numOfProcessesArrived; i++) {

                processes.get(i).setScore(0);
                if (processes.get(i).getProcessCompleted() == true) {
                    processesCompleted += 1;
                    totalBurstsCompleted += processes.get(i).getActualBurstTime();
                };

            }

            for (int i = 0; i < numOfProcessesArrived; i++) {

                for (int j = 0; j < numOfProcessesArrived; j++) {

                    int k = j + 1;

                    if(k == numOfProcessesArrived) {
                        k = numOfProcessesArrived - j;
                    }

                    if(processes.get(i).getArrivalTime() <= currentLoopIteration) {

                        if(processes.get(i).getExpectedBurstTime() <=
processes.get(k).getExpectedBurstTime()) {
                            processes.get(i).addScore(1);
                        }

                        if(processes.get(i).getPriority() < processes.get(k).getPriority())
{
                            processes.get(i).addScore(numOfProcessesArrived);
                        }

                        if(processes.get(i).getWorked() >
processes.get(i).getExpectedBurstTime()) {
                            processes.get(i).addScore(numOfProcessesArrived * 2);
                        }

                        if(processesCompleted != 0) {
                            if(processes.get(i).getAge() > (totalBurstsCompleted /
processesCompleted)) {        // Average burst time of all the completed processes
                                processes.get(i).addScore(numOfProcessesArrived * 4);
                            }
                        }
                    }
                }
            }

            if (numOfProcessesArrived != 0) {
                System.out.print("Current Time: " + (currentLoopIteration) + "\n");
                displayTitles();
            }

            for (int i = 0; i < numOfProcessesArrived; i++) {
                processes.get(i).displayProcess();
            }

            if (numOfProcessesArrived != 0) {
                System.out.print("\n");
            }

            numOfProcessesArrived += 1;

            int positionScore1 = 0;
            int positionScore2 = 0;
            int positionScore3 = 0;
            int positionScore4 = 0;

            @SuppressWarnings("unused")
            int positionScore5 = 0;

            if (numOfProcessesArrived <= totalProcesses) {
                int pastLoopIteration = currentLoopIteration;
                currentLoopIteration = processes.get(numOfProcessesArrived-
1).getArrivalTime();
                int timeLeft = currentLoopIteration - pastLoopIteration;

                int positionSlots = 1;

                while (positionSlots <= numOfProcessesArrived) {

                    position1 = process1;
                    positionScore1 = process1.getScore();

                    positionSlots += 1;

                    if (process2.getScore() > positionScore1) {

                        position2 = position1;
                        positionScore2 = positionScore1;

                        position1 = process2;
                        positionScore1 = process2.getScore();

                    }
                    else {

                        position2 = process2;
                        positionScore2 = process2.getScore();

                    }

                    positionSlots += 1;

                    if (process3.getScore() > positionScore1) {

                        position3 = position2;
                        positionScore3 = positionScore2;

                        position2 = position1;
                        positionScore2 = positionScore1;

                        position1 = process3;
                        positionScore1 = process3.getScore();

                    }
                    else if (process3.getScore() > positionScore2) {

                        position3 = position2;
                        positionScore3 = positionScore2;

                        position2 = process3;
                        positionScore2 = process3.getScore();

                    }
```

```
else {

    position3 = process3;
    positionScore3 = process3.getScore();

}

positionSlots += 1;

if (process4.getScore() > positionScore1) {

    position4 = position3;
    positionScore4 = positionScore3;

    position3 = position2;
    positionScore3 = positionScore2;

    position2 = position1;
    positionScore2 = positionScore1;

    position1 = process4;
    positionScore1 = process4.getScore();

}
else if (process4.getScore() > positionScore2) {

    position4 = position3;
    positionScore4 = positionScore3;

    position3 = position2;
    positionScore3 = positionScore2;

    position2 = process4;
    positionScore2 = process4.getScore();

}
else if (process4.getScore() > positionScore3) {

    position4 = position3;
    positionScore4 = positionScore3;

    position3 = process4;
    positionScore3 = process4.getScore();

}
else {

    position4 = process4;
    positionScore4 = process4.getScore();

}

positionSlots += 1;

if (process5.getScore() > positionScore1) {

    position5 = position4;
    positionScore5 = positionScore4;

    position4 = position3;
    positionScore4 = positionScore3;
```

```
        position3 = position2;
        positionScore3 = positionScore2;

        position2 = position1;
        positionScore2 = positionScore1;

        position1 = process5;
        positionScore1 = process5.getScore();

    }
    else if (process5.getScore() > positionScore2) {

        position5 = position4;
        positionScore5 = positionScore4;

        position4 = position3;
        positionScore4 = positionScore3;

        position3 = position2;
        positionScore3 = positionScore2;

        position2 = process5;
        positionScore2 = process5.getScore();

    }
    else if (process5.getScore() > positionScore3) {

        position5 = position4;
        positionScore5 = positionScore4;

        position4 = position3;
        positionScore4 = positionScore3;

        position3 = process5;
        positionScore3 = process5.getScore();

    }
    else if (process5.getScore() > positionScore4) {

        position5 = position4;
        positionScore5 = positionScore4;

        position4 = process5;
        positionScore4 = process5.getScore();

    }
    else {

        position5 = process5;
        positionScore4 = process5.getScore();

    }

    positionSlots += 1;

}
```

```
                for (int i = 0; i < numOfProcessesArrived; i++) {

                    int positionID = positions.get(i).getID();

                    if (timeLeft > positions.get(i).getActualBurstTime() -
positions.get(i).getWorked()) {

                        int timeSpent = positions.get(i).getActualBurstTime() -
positions.get(i).getWorked();

                        switch (positionID) {

                            case 1:
                                process1.setWorked(process1.getActualBurstTime());
                                break;
                            case 2:
                                process2.setWorked(process2.getActualBurstTime());
                                break;
                            case 3:
                                process3.setWorked(process3.getActualBurstTime());
                                break;
                            case 4:
                                process4.setWorked(process4.getActualBurstTime());
                                break;
                            case 5:
                                process5.setWorked(process5.getActualBurstTime());
                                break;
                        }

                        timeLeft -= timeSpent;

                    }
                    else {

                        switch (positionID) {

                        case 1:
                            process1.setWorked(process1.getWorked() + timeLeft);
                            break;
                        case 2:
                            process2.setWorked(process2.getActualBurstTime() + timeLeft);
                            break;
                        case 3:
                            process3.setWorked(process3.getActualBurstTime() + timeLeft);
                            break;
                        case 4:
                            process4.setWorked(process4.getActualBurstTime() + timeLeft);
                            break;
                        case 5:
                            process5.setWorked(process5.getActualBurstTime() + timeLeft);
                            break;

                        }

                        timeLeft = 0;

                    }
                }
            }
            else {

                int sumOfActualBursts = 0;
                for (int j = 0; j < totalProcesses; j++) {
                    sumOfActualBursts += processes.get(j).getActualBurstTime();
                }

                currentLoopIteration = processes.get(numOfProcessesArrived-
2).getArrivalTime() + sumOfActualBursts;

            }

            if (numOfProcessesArrived - 1 == totalProcesses) {
                algorithmRunning = false;
            }
        }
    }

    public static ArrayList<Process> readProcessesFromCSV(String fileName) {

        ArrayList<Process> processList = new ArrayList<>();
        Path pathToFile = Paths.get(fileName);

        try (BufferedReader br = Files.newBufferedReader(pathToFile,
StandardCharsets.US_ASCII)) {

            String line = br.readLine();    // This reads the csv file line by line.
            int processID = 1;
            while (line != null) {

                String[] attributes = line.split(",");
                Process input = new Process(processID,     // The process ID
                        Integer.parseInt(attributes[0]),  // The arrival time
                        Integer.parseInt(attributes[1]),  // The expected burst time
                        Integer.parseInt(attributes[2]),  // The actual burst time
                        Integer.parseInt(attributes[3]),  // The priority
                        false,0,0,0);

                processList.add(input);
                processID += 1;
                line = br.readLine();

            }

        } catch (IOException ioe) {
            ioe.printStackTrace();
        }

        return processList;
    }
}
```

Process Class

```
package dataPackage;

public class Process {

    private int ID;                         // A unique ID for the process
    private int arrivalTime;                // A arrival time of the process
    private int expectedBurstTime;          // The expected burst time of the process
    private int actualBurstTime;            // The actual burst time of the process
    private int priority;                   // The priority of the process
    private String priorityString;          // Converts the priority number into a string
    private boolean completed;              // Checks if the process has been completed
    private int worked;                     // The amount the process has been worked on
    private int age;                        // The age of the process
    private int score;                      // The score of the process

    public Process(int id, int arrival, int expectedBurst, int actualBurst, int
processPriority, boolean processCompleted, int amountWorked, int processAge, int
processScore) {

        ID = id;
        arrivalTime = arrival;
        expectedBurstTime = expectedBurst;
        actualBurstTime = actualBurst;
        priority = processPriority;
        completed = processCompleted;
        worked = amountWorked;
        age = processAge;
        score = processScore;

        if (priority == 1) {
            priorityString = "High";
        }
        else if (priority == 2) {
            priorityString = "Medium";
        }
        else {
            priorityString = "Low";
        }

    }

    public void addScore(int processScore) {
        score += processScore;    // Adds to the score of the process.
    }

    public void displayProcess() {  // This is the displayed structure for each ticket.

        System.out.print(String.format("%1s %1s %2s %1s %7s %6s %11s %10s %7s %2s %4s %2s
%7s %7s %1s", "|", "Process", ID, "|", arrivalTime, "|", expectedBurstTime, "|",
priorityString, "|", score, "|", worked, "|", "\n"));

    }

    public int getActualBurstTime() {
        return actualBurstTime;     // Returns the actual burst time of the process.
    }
    public int getAge() {
        return age;                 // Returns the age of the process.
    }

    public int getArrivalTime() {
        return arrivalTime;         // Returns the arrival time of the process.
    }

    public int getExpectedBurstTime() {
        return expectedBurstTime;    // Returns the expected burst time of the process.
    }

    public int getID() {
        return ID;                  // Returns the process ID.
    }

    public int getPriority() {
        return priority;            // Returns the priority of the process.
    }

    public boolean getProcessCompleted() {
        return completed;           // Returns if the process has been fully completed.
    }

    public int getScore() {
        return score;               // Returns the score of the process.
    }

    public int getWorked() {
        return worked;              // Returns how much of the process has been completed.
    }

    public void setAge(int processAge) {
        age = processAge;           // Sets the age of the process.
    }

    public void setProcessCompleted(boolean processCompleted) {
        completed = processCompleted;       // Sets the process to be completed.
    }

    public void setScore(int processScore) {
        score = processScore;       // Sets the score of the process.
    }

    public void setWorked(int amountWorked) {
        worked = amountWorked;      // Sets how much the process has been completed.
    }

}
```

Reflection

Upon reflection of my work for the three previous tasks, I feel that my knowledge of assessing how the choice of data structures and algorithm design methods impact the performance of a program. I have chosen appropriate data structures and algorithm design methods in order to create effective solutions, utilising object-oriented design principles where possible. I have also demonstrated an understanding of scheduling, designing an effective solution for managing multiple processes.