Sheffield Hallam University

Department of Computing

Algorithms and Data Structures (55-508226-AF-20212)

Name: Joe Kirkup
Student ID: 29026253

Table of Contents

1. Project 1

1.1 Algorithms in ADL

Algorithm to populate an individual row:
FOR number_of_columns
    left = row_above(row_number – 1)
    right = row_above(row_number)
    this_value = left + right
    this_row(row_number) = left + right

1.2 Software and its presentation, including testing (and video link)

The codebase for my Pascal's Triangle project can be found attached to the uploaded assignment.
The video demonstration of the project is available here: https://youtu.be/0jLjszHLFig

1.3 Descriptive Report, including artefacts

1.4.1 Transitioning algorithms to implementation

Before writing the C# implementation of my solution, I produced a basic algorithm in ADL expressing the method for populating each row of the triangle. By this point, I had already envisioned the general approach I was going to use, and everything beyond the ADL segment was trivial.

1.4.3 Problem-solving strategy

The problem of producing and displaying Pascal's Triangle was quite a simple one.
For this reason, I did not need to employ any advanced problem-solving strategies. I did write some basic pseudocode for the algorithm I used to populate individual rows, but beyond that, I would have found it more difficult to complete the project had I completed excessive planning etc. before writing the code.

1.4 Incorporation of formative feedback

Unfortunately, I did not get any formative feedback for this project.

2. Project 2

2.1 Descriptive report, including artefacts

2.1.1 Time-Space Trade-off Choices made

1. Find the origin and destination of a flight, given the flight number

   The data is already stored in a fairly ideal way to do this; the process of execution could be sped up by storing the name of the origin and destination city directly in the table, but this would take up a lot more storage space and be far less maintainable/manageable due to repetition of data as opposed to referencing unique items many times (using relationships).

2. Given city A and city B, find whether there is a flight from A to B, and if there is, find and return its flight number

   One way to achieve this faster would be to have a table for each city which lists departures from there; then one could simply filter for the destination and look at the flight number. Such a table could look like this:

Departures from Boston	
Destination	Flight Number
3	701
5	711
4	746

Again, while this would reduce the execution time of certain operations, a massive amount of storage would be required – 2 extra cells just for every flight. In a system of thousands of flights a day, this would quickly amount to a lot of storage used for little increase in execution speed.

2.1.2 Computational Complexity

The complexity of the algorithm is n^3.

If we look at the algorithm, we see three nested for-loops. If we name the loops *a*, *b*, and *c* from the bottom up and the code inside the last loop *x* then we can describe the running time of the algorithm segment as such:

$(((x_t * a_t) * b_t) * c_t)$

Since we are describing the complexity, and the block of code nested inside the loops is of constant complexity and not dependent on the ever-exponentiating for loop conditions, we can discard *x*;

$((a_t) * b_t) * c_t)$

Since the number of iterations of *a* depend on the current iterative value of *b*, and likewise between *b* and *c*, we can say that in the context of describing the complexity, the total number of iterations is *n * n * n*, because *n* determines the number of iterations for loop *c* and the other loops are dependent on that. Conclusively, n * n * n is more simply written as $\mathbf{n^3}$.

2.2 Incorporation of formative feedback

Unfortunately, I did not get any formative feedback for this project.

3. Project 3

3.1 Algorithms in ADL

3.2 Data Structures

The main data structure used in this project is a linked list, as it allows for random-access insertion and deletion from the queue while offering efficient algorithms in terms of computational complexity in doing so.

3.3 Software and its Presentation, including testing (and video link)

The codebase for my Priority Queue project can be found attached to the uploaded assignment.
The video demonstration of the project is available here: https://youtu.be/HHjBBlEVJns

3.4 Descriptive Report, including artefacts

3.4.1 Transitioning algorithms to implementation

3.4.2 Problem-solving strategy

The first thing I had to do to solve this problem was create a functioning linked list, which I did using the Node and Queue classes seen in my project.
I made sure this worked properly as a simple linked list before adding any functions or code related specifically to the project brief – this ensured that I would have a properly functioning base to my code, and that would give me greater confidence in debugging if I had any issues fulfilling the project requirements.

This general 'stepped' approach is a good one – do not take on multiple tasks at once, split up the problem where possible and debugging the issues you inevitably encounter at each step will become a much less painful process.

This approach did benefit me, because I had several bugs with my linked list implementation that I had to iron out before addressing the specifics of the project brief – both the AddNode and RemoveNode functions seemed at first to work properly, but corrupted the linked list under certain circumstances, such as adding a new node at the head of the list and removing a lower priority node near the bottom of the list.
Due to my segmented approach of the problem I was able to tackle and debug these issues successfully with not much confusion or wasted time debugging code that didn't have any problems.

3.5 Incorporation of formative feedback

Unfortunately, I did not get any formative feedback for this project.

4. Project 4

4.1 Proposed Algorithms

Before beginning to code anything for this project, I sat and thought for a while about what would make an effective algorithm for an elevator system, balancing two things:

- Speed in completing all level requests
- Avoiding leaving certain level requests unvisited for unreasonably long

I soon realised that directly addressing each level request in order is not the most efficient approach, and instead a higher-level solution should work better.

The idea I had was to take the first (oldest) level request still in the queue at any given time, and to start heading towards that level. At each floor on the way, the elevator will check if the current floor matches any other level requests. If it does, the elevator stops and opens the doors, then continues to head to the earlier specified 'target' level. When it reaches that level, the process repeats.

4.1.1 Elevator Algorithm

```
elevatorWaitTime = 0;
System.out.println("The current floor is: "+currentLevel);
int target = floors.getFirst();
while (!floors.isEmpty()/* || currentLevel != 5*/) {
        moveElevator(target-currentLevel);
        if (floors. Contains(currentLevel)) {
                System.out.println("\nStopping and opening doors at level: " + currentLevel);
                floors.removeFirstOccurrence(currentLevel);
                target = !floors.isEmpty() ? floors.getFirst() : currentLevel;
                elevatorWaitTime += 2000;
        }
}
```

4.1.2 The Elevator Algorithm in ADL

```
System.out.println("The current floor is: "+currentLevel);
target = CALL getFirstRequest
WHILE (NOT(floors.isEmpty))
        CALL moveElevator(target-currentLevel)
        IF floors.Contains(currentLevel)
                OUTPUT "Stopping and opening doors at level: ", currentLevel
                floors.removeFirstOccurrence(currentLevel)
                target = CALL getFirstRequest
        END IF
END WHILE
```

4.2 Software and its Presentation, including testing (and video link)

The codebase for my Priority Queue project can be found attached to the uploaded assignment.
The video demonstration of the project is available here: https://youtu.be/tIh0vn0Z4Fo

4.3 Descriptive Report, including artefacts

4.3.1 Experiment Strategy

Using a freshly generated dataset (having just ran GenerateDataset.java), I ran ElevatorExperiment.java twice, once for the example FIFO algorithm and once for my solution, for each size of dataset. The results (total, average, and minimum runtime) are visible below.

4.3.2 Experiment Results

N	FIFO		Algo_Joe	
	Mean	Min	Mean	Min
5	20.3	14.0	19.3	16.0
10	47.5	40.0	40.7	34.0
20	93.8	81.0	80.3	66.0
25	115.4	103.0	95.1	85.0
30	139.4	119.0	109.1	101.0
40	188.8	174.0	141.8	130.0
50	229.3	211.0	174.7	167.0

4.3.3 Findings and Discussion

As you can see, in one case – the minimum time with the smallest dataset – the FIFO algorithm beats out my solution. However, in every other measure, mine is significantly faster, especially in the larger (more realistic) datasets.

I take this as conclusive evidence that the algorithm I designed sufficiently outperforms FIFO, due primarily to its greater understanding of the problem domain – rather than 'dumbly' taking an input of numbers and cycling through them, it has been created with the ability to respond to inputs in a way that improves its effectiveness without any extra computational complexity.

4.4 Incorporation of formative feedback

Unfortunately, I did not get any formative feedback for this project.

5. Project 5

5.1 Algorithms

5.1.1 Introduction to Genetic Algorithm

5.1.2 Proposed Genetic Algorithm in ADL

5.2 Software and its Presentation, including testing (and video link)

The codebase for my Priority Queue project can be found attached to the uploaded assignment.
The video demonstration of the project is available here: https://youtu.be/-oZYbymXhHM

5.3 Descriptive Report, including artefacts

5.3.1 Experiment Strategy

5.3.2 Experiment Results

```
Final gene:
0 0 0 1 0 0 1 0 1 0 1 1 1 1 0 0 0 0 0 1 0 1 1 1 0
Final fitness: 0.7180000000000177
```

5.3.3 Findings and Discussion

5.4 Incorporation of formative feedback

Unfortunately, I did not get any formative feedback for this project.