AVAMAE final interview problem

Josh Tate

May 2022

The source code is available at **github.com/JoshuaTate/AVAMAE-lift-problem**

Contents

1 Introduction and recap

Following the mysterious (and rather convenient) illness sweeping the juniors, I've been asked to write the code I previously discussed purely as general thoughts and pseudo-code in the last stage of the AVAMAE interview process. I'll try not to repeat myself too much as there was quite a lot of detail in the previous report, which hopefully you have to hand.

Our general idea for the lift system for the MedicineChest - which we recall was a single, 10-story lift with only a single, non-directional call button on each floor - was to calculate a "PF" (person-floors) value for every possible lift schedule and pick the minimum. In other words, we want our lift to stop at each floor it is called to (either to pick someone up or drop them off) in an order such that the time spent waiting and travelling in the lift for all the passengers is the least. It's also necessary to place some extra criteria on this to avoid any individual getting stuck in the lift forever, or the lift reaching maximum capacity.

Now, we have to implement such an algorithm in actual code, and test it using the sample data provided by AVAMAE. Said sample data is a CSV file containing a row each time the lift is called, with columns corresponding to the ID of the person calling the lift, the floor they're calling from, the floor they want to go to, and the time at which they call the lift. This simulation should be written in C#, and should return another CSV with a format equal, or at least similar to, a row for each stop the lift makes and columns corresponding to the current time, the people in the lift, the current floor and the floors in the lift's call queue, in order.

1.1 Limitations and possible mistakes

As I'm relatively inexperienced with C#, this was no small task - and given it had to be fit in between work and other social commitments, I'm aware this program won't be perfect! I therefore feel it is best to acknowledge and discuss any mistakes or limitations to this program that I'm aware of before we get into the details.

My first note is that the brief requests that the prototype should be a simple, console application. I have technically produced such a thing, but only because Visual Studio Code (what I used to develop this) automatically produces an executable file representing a console application as part of building the program. My creation is limited to a single (long, but still single) C# file that does not have any GUI code or have the ability to process any user input, so I can't really claim to have produced a console application under my own steam. If this was a requirement, then I apologise for my misunderstanding - and I'd be happy to develop such a thing given a bit more time if desired!

The next limitation of my particular algorithm, as discussed in the previous application stage's report, is that I have made us of a very thorough but very computationally wasteful algorithm. The number of loops the program performs each time it evaluates the lift's future schedule is on the order of the number of calls in the lift's call queue factorial. Using the sample data, the largest call queue has 9 stops, which is 362,880; not a problem on a modern computer, but any possible increase in queue size from this may reach intolerable levels for the kind of budget processor one might use for a lift system. Ways around this were also discussed in the previous report, although since I tailored my algorithm to the dataset provided (and was thus aware my longest queue would be 9), I made no effort to incorporate such process-power-saving methods in this algorithm.

The final significant point to make involves an ambiguity with the given brief - it states that "For the purposes of the prototype, assume the lift takes 10 seconds to move from one floor to the next". It's unclear to me whether this means the lift's velocity is 1 floor every 10 seconds, or if it takes 10 seconds to move from it's current floor to whatever the next floor in it's schedule is. I have developed a solution for each of these, but commented out code relating to the former; it became clear that a lift moving 1 floor every 10 seconds would be prohibitively slow, so I chose to adopt the latter for my final solution.

While hopefully only a minor point, as the program does work in it's current state, I haven't had the time to implement any error handing to catch and manage exceptions; it's therefore possible that changing any of the simulation parameters - even the ones designed to be fiddled with - or using a different dataset will cause the program to crash without any helpful error messages being produced. Innocent mistakes such as making a typo in the file path constants will also crash the program.

2 Documentation

Below is the documentation of all classes, methods and fields used in the algorithm.

2.1 "Init" class

Class containing the "Main" method to launch the C# program; launches both the main lift algorithm and a series of tests beforehand to ensure all helper classes and methods are working as planned.

2.1.1 Fields

public static string **data_source_filepath**:

- The file path for the sample data for the lift simulation to run on, which for me was the "Cloud Software Engineer Coding Exercise Data" CSV file.
- **Should be changed by the user to point at the relevant sample data.**

public static string **output_path**:

- The file path for the CSV output file, containing the results of the simulation
- **Should be changed by the user to point towards the desired destination.**

public static string **debug_path**:

- The file path for the text file which debug statements are dumped into, used extensively by me during debugging
- **Should be changed by the user to point towards the desired destination.**

2.1.2 Methods

public static void **Main**():

- Conducts a series of tests to ensure all helper classes and methods are working (using the *tests* method in the same class).
- launches the main algorithm (by creating a *MedChestLift* object).

public static void **tests**():

- Tests the majority of the support methods used in the algorithm, such as those in the "Utility" class as well as the "LiftObject" methods.
- Prints the result to console so the user is certain the application is functioning correctly.

2.2 "MedChestLift" class

Class containing the lift algorithm logic and the simulation loop, and can be considered as the core of the application.

2.2.1 Fields

public static short **max_individual_PF_travelling**:

- A constant representing the maximum amount of floors a passenger should travel past in the lift before being delivered to their floor, regardless of if it's the most efficient choice.
- *Does not need to be changed by the user, but can be for experimentation purposes.*
- recommended range is 10-20.

public static short **max_individual_PF_waiting**:

- A constant representing the maximum amount of floors a passenger should wait for the lift to travel past before it picks them up, regardless of if it's the most efficient choice.
- *Does not need to be changed by the user, but can be for experimentation purposes.*
- recommended range is 10-20.

public static short **algorithm_start_floor**:

- A constant representing the floor the lift is on at the start of the simulation.
- *Does not need to be changed by the user, but can be for experimentation purposes.*
- Has to be between 1 and 10 inclusive.

2.2.2 Methods

public **MedChestLift**():

- The class constructor.
- Creates a "CSVDataFrame" object to be populated with the CSV file at *Init.data_source_filepath*; not having the proper CSV will crash the program.
- Creates the *LiftObject* object and launches the main algorithm by calling it's own *mainAlgorithm* method.

public void **mainAlgorithm**(*CSVDataFrame* **data**, *LiftObject* **lift**):

- Tests the proposed lift algorithm by testing it in a simulation using the sample data provided by **data**.
- The commented source code will likely provide a better explanation than offered here, but essentially works by ticking forward in time by one increment and applying the algorithm logic every time the lift moves floor or receives a new call.
- Will sleep for 60 seconds at the end of the simulation before closing so anyone running the executable file on its own instead of directly running the source code through an IDE will get a minute to read the results; I recommend removing this line (line 246) if you intend to only run the source code directly through an IDE.

2.3 "LiftObject" class

Class intended to be used as the "lift object". Stores the present state of the lift and includes a number of helper methods to the main algorithm.

2.3.1 Fields

public static short **num_floors**:

- A constant representing the number of floors in the simulated building; unlike an array in C#, the convention is that the floors start from 1.
- Is not actually used at any point as the maximum and minimum floors are inferred from the sample data.

public static short **max_capacity**:

- A constant representing the maximum amount of people who can be in the lift at any one time.
- *Does not need to be changed by the user, but can be for experimentation purposes. Note that the design brief specifies that this constant should be 8.*
- **Since the sample data does not cause this to "bite" at any point, at least using my algorithm, the logic that implements this maximum remains untested due to time constraints.**

public static short **time_per_floor**:

- A constant representing the time taken for the lift to move between floors
- *Does not need to be changed by the user, but can be for experimentation purposes. Note that the design brief specifies that this constant should be 10.*
- Changing this to a higher number may cause the program to stall due to increased schedule lengths, which the program cannot realistically deal with due to the method by which it finds the most efficient schedule.

public Dictionary[short, LiftPassenger] **passengers**:

- A dictionary of short keys with corresponding *LiftPassenger* object values; used to store additional data about lift passengers.

public object[] **lift_schedule**:

- The schedule to be populated once the lift algorithm starts running.
- elements have the convention *(ID)_(Floor)("D"/"C")* where (ID) is the integer ID of the passenger, (Floor) is the integer of the floor the lift has been called to and "D"/"C" represent the lift being called to drop a passenger off or pick them up respectively. For example, passenger 5 asking to be dropped off at floor 7 would be *5_7D*.

public short **curr_floor**:

- A 16-bit integer used to store what floor the lift is currently on.

public short **curr_direction**:

- A 16-bit integer used to store what direction the lift is currently moving in.
- *This is a legacy field originally used in the system where the lift moves at 1 floor per 10 seconds, and currently is unused as the relevant code blocks in the main algorithm have been commented out.*

2.3.2 Methods

public **LiftObject**(*short* **start_floor**):

- The class constructor.
- Sets the *curr_floor* of the lift to the *start_floor*, which is where the lift will be at the start of the simulation. Currently does not have a default value so needs to be passed this as a parameter.

public object[] **getScheduleThatMinimisesPF**():

- Returns an *object[]* with the same convention as the *lift_schedule* field above.
- Computes every possible schedule the lift could have for the existing calls, evaluates if each schedule is valid given criteria imposed by *MedChestLift.max_individual_PF_travelling*, *MedChestLift.max_individual_PF_waiting* and *LiftObject.max_capacity.*
- Returns the schedule with the lowest calculated *PF* value.

public int **calcSchedulePF**(*object[]* **schedule**):

- Calculates the *PF* value for a given Schedule.
- Will also refer to the *LiftObject* instance it is called from to ensure it does not breach the criteria imposed by *MedChestLift.max_individual_PF_travelling* and *MedChestLift.max_individual_PF_waiting*, returning -1 if it does.

public static short **schedID**(*object[]* **sched**):

- Returns the 16-bit integer of the passenger's ID given their entry from the *lift_schedule*, as in the convention described above.

public static short **schedFloor**(*object[]* **sched**):

- Returns the 16-bit integer of the passenger's target floor given their entry from the *lift_schedule*, as in the convention described above.

public static short **schedType**(*object[]* **sched**):

- Returns the char representing the passenger's call type (e.g. a call to be picked up by the lift from a floor or a call to be dropped off by the lift given they are already inside) given their entry from the *lift_schedule*, as in the convention described above.

2.4 "LiftPassenger" class

Class intended to be used as the "passenger object" for each passenger throughout the simulation. Stores the present state of the passenger to provided more detail about their journey when required.

2.4.1 Fields

public short **origin_floor**:

- Stores the floor the passenger called the lift from.

public short **destination_floor**:

- Stores the floor the passenger requested the lift drop them off at.

public short **waiting_PF**:

- Stores the *PF* value for the time spent waiting for the lift to pick the passenger up.

public short **travelling_PF**:

- Stores the *PF* value for the time spent travelling in the lift.

public short **origin_time**:

- Stores the time when the person originally called the lift.

public short **total_time**:

- Stores the total time taken for the passenger to complete their journey.

public bool **in_lift**:

- Describes whether the given passenger is currently in the lift or not.

public bool **journey_completed**:

- Describes whether the given passenger has completed their journey or not.

2.4.2 Methods

public **LiftPassenger**(*short* **origin**, *short* **destination**, *short* **time**):

- The class constructor.
- Sets the object's *origin_floor*, *destination_floor* and *origin_time* fields to *origin*, *destination* and *time* arguments respectively.
- Does not have any default arguments so all 3 need to be passed to the method.

public string **debugString**():

- Returns a convenient string containing all of the fields and their values for a particular instance.

2.5 "CSVDataFrame" class

An object intended to be populated with CSV-esque data for easy manipulation, loading and storage of such data.

2.5.1 Fields

public string[] **column_names**:

- Stores the column names for the data frame.
- Should not be accessed directly by the user, and column names should be set through the class constructors.
- *Ideally should be encapsulated or set to private but was not due to debug convenience and time constraints.*

public string[][] **column_data**:

- Stores the column data for the data frame.
- Should not be accessed directly by the user, and column data should be set through the class constructors and *appendToColumn* method.
- *Ideally should be encapsulated or set to private but was not due to debug convenience and time constraints.*

2.5.2 Methods

public **CSVDataFrame**(*string* **filepath**):

- The class constructor - overloaded to take a single string argument corresponding to the file path of a CSV file.
- Will load the CSV file's data and construct a data frame using this data.

public **CSVDataFrame**(*string[]* **new_column_names**):

- The class constructor - overloaded to take a single string[] argument corresponding to the names of the columns in a desired new data frame.
- Will create an empty, template data frame using the inputted column names.

public void **constructDataFrameFromCSV**(*string* **filepath**):

- Constructs the data frame from a CSV whose location is given by *filepath*.
- Is called automatically by the relevant class constructor so should not ever need to be called manually by the user.

public void **constructDataFrameManually**(*string[]* **new_column_names**):

- Constructs an empty template data frame using column names as given by *new_column_names*.
- Is called automatically by the relevant class constructor so should not ever need to be called manually by the user.

public static string[] **readCSV**(*string* **filepath**):

- Reads the CSV at *filepath* and returns it as a simple array with each line as raw text.
- Does not contain exception handling so the *filepath* needs to be correct or the program will crash.
- Used by the class constructor for this object, but can also be used as a utility function by the user.

public void **printTable**():

- Prints the current contents of the data frame to the console in a conveniently readable format.
- useful in debugging, and is called at the end of the main lift algorithm to show the lift's process.

public object[] **column**(*string* **name**):

- Returns the column of the data frame given it's name.

public object[] **appendToColumn**(*string* **name**, *object* **item**):

- Appends the *item* to the relevant column given that columns name.

public int **length**():

- Simply returns the length of the data frame.
- Naively takes the length of the 0th column, so will only reliably work in instances where the data frame has all columns as the same length.

public void **saveCSV**(*string* **path**):

- Saves the current contents of the data frame to a CSV file at *path*.

2.6 "Utility" class

An helper class containing a number of generic utility methods to assist with the lift algorithm.

2.6.1 Methods

public **Utility**():

- The class constructor
- Has no function, is there purely as the default class template for C#.

public static object[] **subsetArray**(*object[]* **array**, *int* **from**, *int* **to**):

- Returns a slice of the given array, from the *from* index inclusive to the *to* index exclusive.

public static object[] **addArrays**(*object[]* **array1**, *object[]* **array2**):

- Adds the two arrays together into a new, 1D array (e.g. appends all the elements of array2 into array1).

public static object[] **appendArrays**(*object[]* **array**, *object* **element**):

- Appends the *element* to the end of the given *array*.
- Has been overloaded and there is also a version dealing explicitly with strings for convenience when saving files as raw text.
- I'm aware this function is poorly named in hindsight, as it insinuates it appends another array to an existing array but this is what *Utility.addArrays* should be used for.

public static string[] **appendArrays**(*string[]* **array**, *string* **element**):

- Appends the *element* to the end of the given *array*.
- Has been overloaded and there is also a version dealing explicitly with strings for convenience when saving files as raw text.
- I'm aware this function is poorly named in hindsight, as it insinuates it appends another array to an existing array but this is what *Utility.addArrays* should be used for.

public static object[][] **arrayAppendArrays**(*object[][]* **array**, *object[]* **subarray**):

- Appends the 1D array *subarray* to the end of the 2D array *array*.

public static object[] **rotateArray**(*object[]* **array**):

- Rotates the given array - that is, moves all elements down by one index and moves the 0th element to the back of the array.

public static object[][] **getAllPossibleListCombinations**(object[] array):

- Returns a 2D array containing every possible combination of ordering the elements in the given 1D array.
- works on the order of "array length" factorial, so will incur significant delays when used for arrays of length 10 or over.

- Does not contain optimisations for e.g. repeated elements, so *0,0,0,0,0,0,0,1* will take as long to process as *0,1,2,3,4,5,6,7*.

public static List¡object[]¿ **calcListCombs**(*object[]* **source_list**, *List¡object[]¿* **combs**, *object[]* **bolt**):

- Populates the given 2D list *combs* with every possible input of a 1D array given to *Utility.getAllPossibleListCombinations*
- Is not intended to be called directly by the user; it works with *Utility.getAllPossibleListCombinations* so that method should be called to achieve the result of finding all possible unique combinations of an array, not this one.
- Works by recursion; I understand some companies wish to avoid recursion due to it's unpredictable memory usage.

public static void **appendArrayToFile**(*object[]* **array**, *string* **path**):

- Appends a given array to the end of a file located at *path* (or creates the file if necessary) as raw text.

public static void **appendStringToFile**(*string* **data**, *string* **path**):

- Appends a given string to the end of a file located at *path* (or creates the file if necessary) as raw text.

public static void **clearFile**(*string* **path**):

- Creates an empty file, or clears an existing one, at the location of *path*.

3 Maths behind ”getAllPossibleListCombinations”

Previously we have discussed that my lift algorithm works by calculating all possible schedules for the lift, before picking out the best one that satisfies all relevant criteria. However, other than mentioning it works by recursion, and has something to do with rotating the array, I've not discussed in detail exactly how it works. It's a fairly nifty bit of maths if I do say so myself, so I thought it deserved an extra section!

I looked forward in time and saw "N!" ways of ordering the lift's schedule
How many of them satisfied all criteria and were efficient?
Hopefully at least 1

Figure 1: A visual respresentation of how the lift algorithm works, featuring some probably much needed comic relief.

We know from the previous report that there are ”N” factorial unique ways of ordering ”N” objects in a line; the derivation is beyond the scope of this exercise but it's well published online. The problem we need to solve is exactly *how* do you get all ”N” factorial combinations of these elements in a line? Because there isn't a huge amount of desire for brute force algorithms such as this, it's not an easy thing to find online so we're going to have to think of our own algorithm! I'm unsure if there is a better way of doing this, or if I'm independently discovering an existing, popular method, but I decided upon my algorithm in the following way:

Consider an array of two elements, say [0,1]. One trivially sees there are 2 ways of ordering this array - [0,1] and [1,0]. Essentially, we have rotated the array to find both unique combinations.

Now, consider the 3-element array, [0,1,2]. We know there are 3!, or 6, unique combinations for us to find. If we rotate these arrays around a "full cycle", we get [0,1,2], [1,2,0], [2,0,1]. We can see we've found 3 of our 6 unique arrays, but are missing 3; however, if we take each of our 3 arrays, and rotate *all but the very first element*, we find the extra 3: [0,2,1], [1,0,2], [2,1,0]. For example: We take the [0,1,2] array, and we hold the "0" in it's place and then rotate the rest of the array, meaning we bring the "2" forward one element and push the 1 to the back. We have seen that we have found all 6 unique combinations from rotating our array!

As it turns out, for an array of size "N", it is possible to generate every single possible combination of ordering the elements by first rotating the array of size "N", and then for each of these resulting N unique arrays, finding the "N-1" unique rotations of the subset of this array given by removing the first element. For each of the now "N * (N-1)" resulting unique elements, we then once again find the "N-2" unique ways of ordering the subset of each unique array given by removing the first 2 elements, and so on.

If we define a function that calculates all rotations of a size "N" array, we realise we can then have the function strip off the first element of each of these N unique arrays, and call itself with each of them to create a "chain reaction" of sorts that will eventually generate all "N!" unique arrays[1]. Having a function call itself is a popular coding method known as recursion, and that's what the method *Utility.calcListCombs* does.

[1]The function in practise actually generates several duplicates - I'm not sure why, but as it turns out as "N" tends to infinity, the number of actual arrays generated by this function tends to N! * e, where e is the natural, or Euler's, number, 2.71828...