## Criterion B: Design

**studentList**

ArrayList<student> studentlist

int [ ][ ] commontime

void add(student)

ArrayList<student> getList( )

boolean empty( )

void setTheArray(int[ ][ ])

 int[ ][ ] getCommonTime

void findStudent(String)

void printStudents( )

ArrayList<int[ ]> counting( )int checkPair(int,int)

**UML**

**student**

int ID

String name

String[14] timesfree

Student(String, String[ ], int IDin)

int getID( )

String getName( )

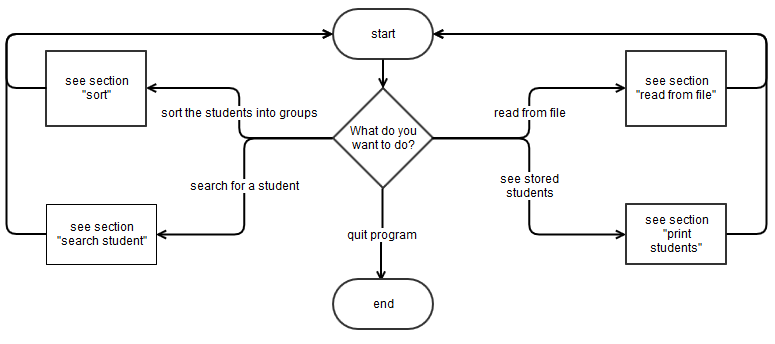
String[ ] getTable( )

void printStudent( )

**timetable**

String[36] studentarray

String[ ] readfromfile()

****

**(¯`·.\_.·(¯`·.\_.· OVERVIEW ·.\_.·´¯)·.\_.·´¯)**

**Contents**

[READ FROM FILE 2](#_Toc440452061)

[SORT 3](#_Toc440452062)

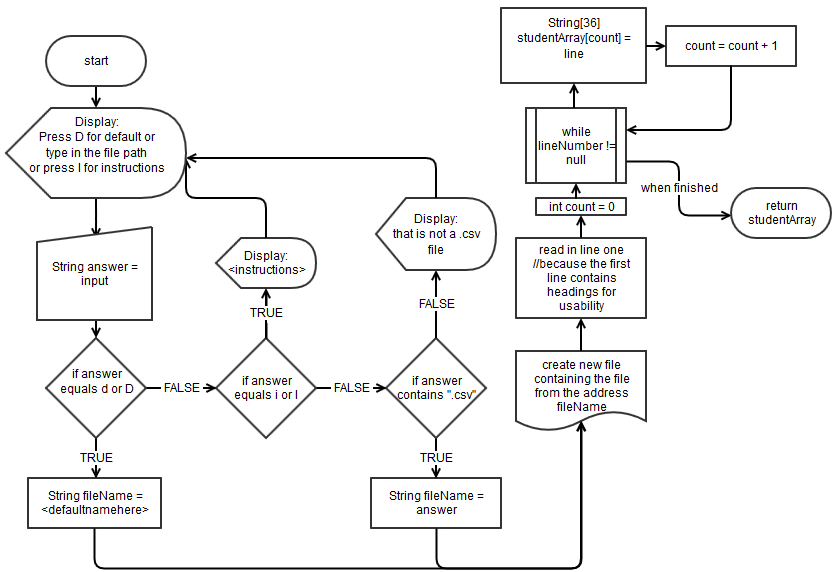
[PRINT STUDENTS 9](#_Toc440452063)

[SEARCH FOR STUDENT 10](#_Toc440452064)

[Test Plan 11](#_Toc440452065)

# (¯`·.\_.·(¯`·.\_.· READ FROM FILE ·.\_.·´¯)·.\_.·´¯)

**(¯`·.\_.·(¯`·.\_.· PAR 1: READ ·.\_.·´¯)·.\_.·´¯)**

****

**(¯`·.\_.·(¯`·.\_.· PART 2: STORE ·.\_.·´¯)·.\_.·´¯)**

int **STUDENTARRAYLENGTH** //this will equal the length of the array

String[] **STUDENTARRAY** //this will have been passed out of the flowchart above

while(**STUDENTARRAYLENGTH** > 0)

store **STUDENTARRAY**[**STUDENTARRAYLENGTH**] //this will involve storing it as an object in

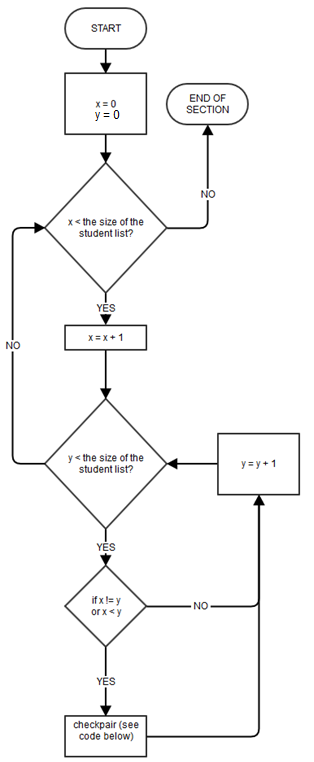
the ArrayList and separating it by the commas

**STUDENTARRAYLENGTH** = **STUDENTARRAYLENGTH** – 1

# (¯`·.\_.·(¯`·.\_.·SORT ·.\_.·´¯)·.\_.·´¯)

Initialize a 2D array (SHAREDTIMES) with parameters equaling the number of students

**(¯`·.\_.·(¯`·.\_.· SELECTING WHICH STUDENTS TO COMPARE ·.\_.·´¯)·.\_.·´¯)**



**(¯`·.\_.·(¯`·.\_.· Checking the pair ·.\_.·´¯)·.\_.·´¯)**

The next stage is to get the timetable of each student which is stored in the student object in the form of a 1D string array (with 15 data addresses, as there are two weeks available to be analysed and one for the name) like the one below:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Student Name | 3 | 5 | 2 | 0 | 0 | 0 | 1 | 3 | 5 | 2 | 0 | 0 | 0 | 0 |

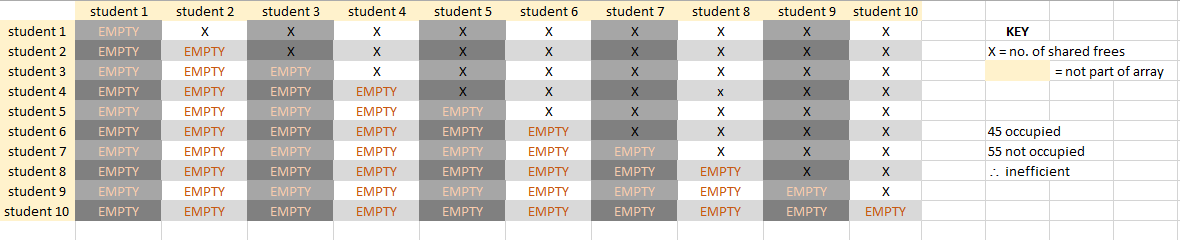
//each location represents a day and each number represents the time on that day they are free (0 = not free)

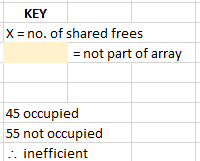
The timetables are then compared to find which one has the most numbers in common. The program will cross-reference each timetable with each other, recording the integer of the number of times they have in common in a 2D array.

For example there are these students:

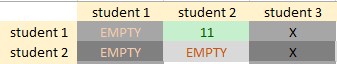
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Student 1 | 3 | 5 | 2 | 0 | 0 | 0 | 1 | 3 | 5 | 2 | 0 | 0 | 0 | 0 |
| Student 2 | 3 | 6 | 2 | 4 | 4 | 0 | 1 | 3 | 5 | 2 | 0 | 0 | 0 | 0 |

Their common times have been highlighted and add up to eleven. This data would be stored in a structure like this (a 2D int array with parameters the size of the number of students):





So, for this specific example above the array would look like this:

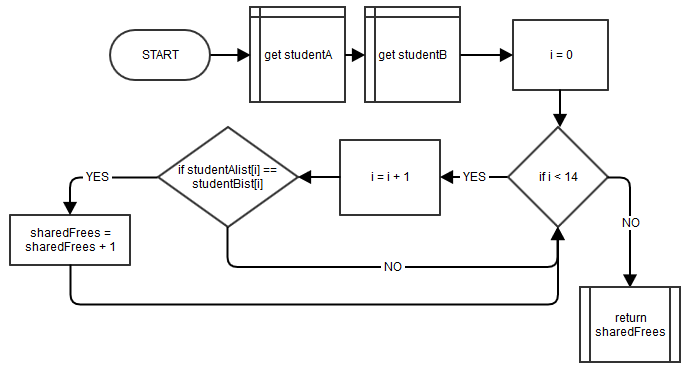


The student headings are not a part of the array, the students are added in their order in the studentList arraylist so they can be easily retrieved later.

As stated above, this use is inefficient as over half are not used. However I corrected the assumption that they would have to be empty as this would call an error when looking through to find the largest number, and instead initialized the array with every data address as “-1”.

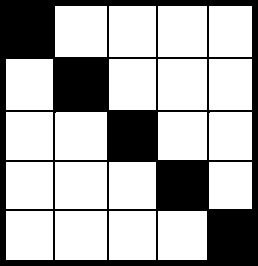
Here is the flowchart for comparing the two int arrays and storing the result in the correct section of the array: (please note that since there is a variable number of

students I have omitted the loops as this would be difficult to communicate in a flowchart, however the whole thing would be repeated for each student pair, based on the length of the arraylist containing the students)

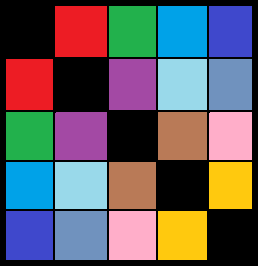


**Side Note on the use of a 2D Array**

The blacked out areas would never be required as they would represent the number of frees a student has with themselves:

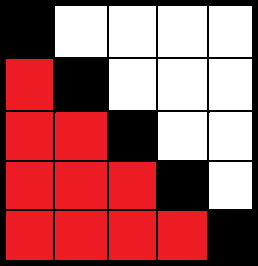
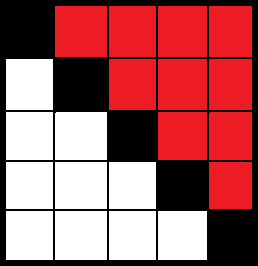
****

However then both halves are duplicates, so here is an image with colours matched where the data would match (so they would be representing the same relationship between two students):



NB: This has a diagonal line of symmetry

So as you can see we only need 45 of 110 squares (41%). It could then be either side:

I chose to fill only the top squares. In order to do this I did each student pair only one each way around, so only compared Student 1 and Student 2, not also Student 2 and Student 1. This saves memory, as the script would have to be executed twice if both were filled.

**(¯`·.\_.·(¯`·.\_.· FIND THE LARGEST NUMBER IN A 2D INT ARRAY ·.\_.·´¯)·.\_.·´¯)**

This code is for once the common frees have been stored in a 2D int array, for finding the two students with the highest number of common frees.

**VARIABLES:**

* int **YCOUNT** = 0 //counting for the x axis
* int **XCOUNT** = 0 //counting for the y axis
* int **BIGGEST** = 0 //record of the biggest value
* int **X** //this represents the length of the ArrayList containing the students - studentlist
* int[] **COORDS** = 2 //to return the index of the biggest value
* int[][]**SHAREDTIMES** = **SHAREDTIMES**[x][x] //the array we sort through

for(**XCOUNT** < **X**; **XCOUNT** = **XCOUNT** + 1) //count through the y axis

for(**YCOUNT** < **X**; **YCOUNT** = **YCOUNT** + 1) //count through the x axis

if(**SHAREDTIMES**[**XCOUNT**][**YCOUNT**] > **BIGGEST**) //if the value in the array is bigger

//than any before it

**BIGGEST** = **SHAREDTIMES**[**XCOUNT**][**YCOUNT**] //it replaces the biggest value

**COORDS**[0] = **YCOUNT** //we store each coordinate so we can trace which

//students it was

**COORDS**[1] = **XCOUNT**

return **COORDS** //return which students (based on their index number) had the largest number of

//shared frees

**TRACE TABLE (if: X = 5, SHAREDTIMES can be seen below in its entirety)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **YCOUNT** | **XCOUNT** | **SHAREDTIMES**[**YCOUNT**][**XCOUNT**] | **BIGGEST** | **COORDS**[0] | **COORDS**[1] |
| 0 | 0 | -1 | 0 | - | - |
| 0 | 1 | 3 | 3 | 0 | 1 |
| 0 | 2 | 4 | 4 | 0 | 2 |
| 0 | 3 | 5 | 5 | 0 | 3 |
| 0 | 4 | 6 | 6 | 0 | 4 |
| 1 | 0 | -1 | 6 | 0 | 4 |
| 1 | 1 | -1 | 6 | 0 | 4 |
| 1 | 2 | 2 | 6 | 0 | 4 |
| … | … | … | … | … | … |
| 4 | 4 | -1 | 6 | 0 | 4 |

**RETURNS: COORDS**

**COORDS[0] = 0; COORDS[1] = 4**

**SHAREDTIMES:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| -1 | 3 | 4 | 5 | **6** |
| -1 | -1 | 2 | 3 | 5 |
| -1 | -1 | -1 | 1 | 1 |
| -1 | -1 | -1 | -1 | 5 |
| -1 | -1 | -1 | -1 | -1 |

The script would identify the biggest number. If there were, in this example, two sixes, it would identify the first biggest number – this does not interfere whatsoever with the functionality of the program.

**(¯`·.\_.·(¯`·.\_.· BLANK OUT THE STUDENTS WE HAVE ALREADY SORTED THROUGH ·.\_.·´¯)·.\_.·´¯)**

It is necessary to blank out the students which have already been sorted so students aren’t put in multiple groups – one student can only be in one group.

**VARIABLES:**

int **COUNT** = 0

int **X** //the size is the length of the ArrayList

int[][] **SHAREDTIMES**= **SHAREDTIMES**[**X**][**X**]

int[] **COORDS** = 2

for(**COUNT** < **X**; **COUNT** = **COUNT** + 1)

**SHAREDTIMES** [**COUNT**][**COORDS**[1]] = -1;

**SHAREDTIMES** [**COORDS** [0]][**COUNT**] = -1;

**SHAREDTIMES** [**COORDS** [1]][ **COUNT**] = -1;

**SHAREDTIMES** [**COUNT**][**COORDS** [0]] = -1;

---------------------------------------------------------------------------------------------------------------------------

if: **X** = 5; **COORDS**[0] = 1; **COORDS**[1] = 2 //SHAREDTIMES has been filled with random numbers

// for illustrative purposes

**SHAREDTIMES** before script run:

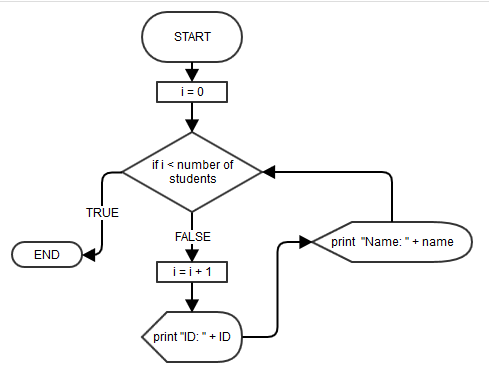
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 8 | 8 | 8 | 8 | 8 | 8 |
| 8 | 8 | 8 | 8 | 8 | 8 |
| 8 | 8 | 8 | 8 | 8 | 8 |
| 8 | 8 | 8 | 8 | 8 | 8 |
| 8 | 8 | 8 | 8 | 8 | 8 |

**SHAREDTIMES** after script run:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 8 | **-1** | **-1** | 8 | 8 | 8 |
| **-1** | **-1** | **-1** | **-1** | **-1** | **-1** |
| **-1** | **-1** | **-1** | **-1** | **-1** | **-1** |
| 8 | **-1** | **-1** | 8 | 8 | 8 |
| 8 | **-1** | **-1** | 8 | 8 | 8 |

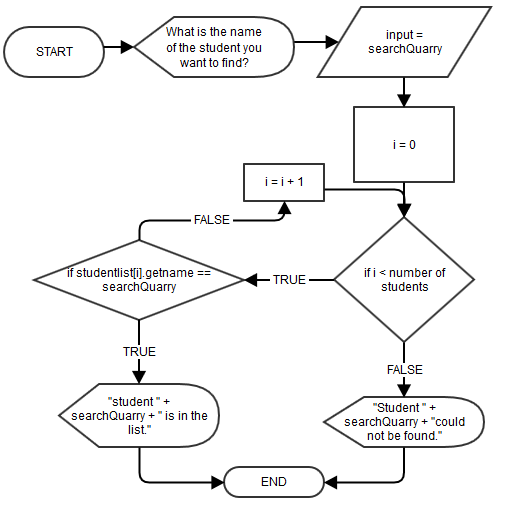
# (¯`·.\_.·(¯`·.\_.· PRINT STUDENTS ·.\_.·´¯)·.\_.·´¯)

This section is relatively simple, I will just have a method in the student object which will allow me to print the student (as its attributes). This is an outline:



# (¯`·.\_.·(¯`·.\_.· SEARCH FOR STUDENT ·.\_.·´¯)·.\_.·´¯)

This algorithm is also simple. Here it is below:

****

# TEST PLAN

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Purpose** | **Description** | **Expected Result** |
| **0** | Read .CSV file | 1. Run program 2. Choose to read from .csv | No error messages – read successful. |
| **1** | Data from file stored correctly in student object(s)  Also “print students” is working | 1. Run program 2. Read from .CSV 3. Choose “Print Students” | List of printed students consistent with those in the file. |
| **2** | Search function 1 | 1. Run program 2. Read from .CSV 3. Search for a student who was in the .CSV file using the “search” option | Student should show up as found. |
| **3** | Search function 2 | Repeat with a different student. | Student should show up as found. |
| **4** | Sorting the students into pairs | 1. Run program 2. Read from .CSV 3. Choose “sort into pairs” 4. Repeat with different numbers and groupings of students | The optimal pairings should be returned. This needs to be repeated with different conditions to cover all eventualities. |
| **5** | The program can run on an Apple OS | 1. Use a shell script to run the program using Terminal 2. Check it still functions | The program should run perfectly, only on the Mac X OS |