University of Liverpool

Computer Science with a Year in Industry BSc (Hons)

G403

COMP390 Honours Year Computer Science Project

Design Specification

Author:
N Aishah B M Senin
(200912462)

Project Advisor:
Dr Prudence Wong

November 17, 2015

Contents

Chapter 1

Overview

1.1 Project description

This project primarily focuses on the animation of different types of commonly used algorithms, for the benefit of users to further understand how algorithms work in general. Learning about what algorithms are and how they work is essential for students who are studying computer science. Since this project is meant to be educational, the target audience of the software will be students studying computer science, or at least have an interest on how computer programs are made efficient. In this project, the scope will revolve around the animations within the three main algorithmic paradigms, such as the greedy method, divide and conquer, and dynamic programming. Also, some of the sorting algorithms as well.

The primary purpose of this project is to develop a software that displays animations which shows how different types of algorithms within the 3 main paradigms and sorting, works in general. From the program, the users are able to pick the algorithmic solution they wish to learn, either enter a certain amount of input or generate random values, and then learn how the algorithm works by watching the animations presented to them.

1.2 Aims and objectives of this project

The primary objective to this project is simply to make difficult algorithms easily understood. To achieve such feat for instance, would be using animations, which acts as a visual aid for the students to learn the algorithms. As a computer science student myself, I believe that using visual tools such as the animations, would certainly enhance the students' learning experience in regards to this topic. This would also allow students to learn new algorithmic problems with convenience and ease.

It is generally known the algorithms is one of challenging topics within the computer science field that is difficult for students to grasp on. In this case, another aim for this project is to make this software as an additional tool, that could be used outside lecture periods, and assist students to achieve greater understanding in algorithmic paradigms. To achieve this, the program is to provide a comprehensive animated explanation of the algorithms in a step by step basis. This strategy of scrutinizing the algorithm allows the users to speculate the complicated algorithms in its granulated state, on how it works in each step, and then making a connection between the sequence of steps that makes the algorithm work as a whole.

Another aim for this project, is to provide the basic idea of how an educational program is suppose to look and work like in order to successfully assist the students. As a computer science student myself, I understand what are the specific difficulties when it comes to learning algorithms, and using them to address every difficulty I had when learning algorithms for the first time. Hopefully, once this project is completed, it will show the other developers who are

interested in taking on this project on the specific areas to pay attention to when developing an educational program like this one.

One of my intentions for this project is to serve this software as foundation. From here, other developers who could use this as a platform, and populate the existing list by many other algorithms and its animations. If the project is deemed successful, universities could use this software to assist other computer science students who are studying algorithms, or generally facing difficulty understanding the concept of them.

Another aim for this project that would be nice to achieve, other than to benefit the students learning process, is to increase the students' interest on this topic. Algorithms is one of my favourite topics I had came across as a computer science student during my course in university. By designing and developing this program, I hope to achieve the same sentiments in regards to my interest in algorithms to other students who are studying this topic as well.

1.3 Summary of research and analysis

Do this

Chapter 2

Design

In this design chapter primarily consists of the plan for the construction of the *Algorithm Animation* program. This design documentation includes the list of requirements that is expected of the program, the different uses of the program from the users' perspective, system flowchart, the graphical user interface design, sequence diagrams, the design of the animations for each algorithms, and finally the evaluation list.

2.1 Brief description of the anticipated software

The program initiates by displaying the main menu. From here, the users can select whichever category they wish to use. These categories include the three main different features of the program, which are *settings*, *animation*, and *appendix*.

With the *settings* feature, the user can change some of the aspects of the program, such as its font size and the speed of animation. This would allow the users to work on an environment that feels most comfortable for them.

The animation feature, which admittedly would be the main aspect of this project, is where the list of different algorithms would be collected at. From here, the user can select the algorithm they wish to learn. Once they have selected one, the program will lead the user to the input page, which has a form for the user to fill. This form is different depending on which algorithm selected, as different algorithms have different sets of input. The user can either enter their own input (these input will be bounded with a specific limit), or generates random values instead.

Once the input page is filled with necessary values, the program proceeds in the animation page, where the animation starts playing from. The anticipated program will also have *controls* for the animation as well, such as the *play*, *pause*, *stop*, and *backtrack* buttons. Each of these controls are described further in depth within this design chapter.

The last feature of this program, is the *appendix*. The appendix would basically display extra information in regards to the algorithms that are not mentioned within the animation feature. The reason for this feature is for students who decided to read further about those algorithms that are presented in the program.

2.1.1 Functional requirements

The functional requirements of the *Algorithm Animation* program is to specify the list of requirements, the system needs to do, such as its behaviour or function. The tables below (tables 2.1 and 2.2) shows the list of functional requirements that is expected for the program to achieve, along with their respective descriptions.

Table 2.1: Functional requirements of the software

No.	Requirements	Description
Menu		
1	Shows the list of playable algorithms	In the menu, the program is to show all the algorithms available in the program in the main list. In this list, user can select whichever algorithm they wish to see.
2	Classify the available algorithms between the 3 main algorithmic paradigms	On the main list, the algorithms are to be classified between the 3 main paradigms, such as the greedy method, divide and conquer, and dynamic programming. This is to allow the users to understand immediately the correlation between similar algorithms when classified within its paradigms. This is also to increase the ease of usability, as users will only be required to look within the algorithms paradigm to search for a specific problem.
Animation		
3	Plays the animation	When the animation is in its initial or paused state, users can play the animation. This initiates the animation, which plays until the end, unless the user either pauses or stops the animation.
4	Pauses the animation	The user can pause the animation, which stops the animation temporarily at its current state.
5	Stops the animation	When the animation is playing, user can stop the animation. This ends the animation completely at any point of time during the playtime of the animation.
6	Backtracks the animation	During the animation's playtime, the program keeps track on the number of iteration(s) the animation is currently at. When a user chooses to backtrack the animation, the animation will *rewind* itself from its current iteration i, to i - *1*.
7	Shows a short description during the animation on each *iteration* of the algorithm	During the animation's playtime, the program is to show a short description about what the animation is doing.

Table 2.2: Functional requirements of the software

No.	Requirements	Description
Help option		
8	Adjust the speed of the animation	Users can adjust the speed of the animation ranging from 1 (very slow), to 10 (very fast). By default, the speed of the animation will be set to 5.
9	Adjust the font size	Users can adjust the font size to fit their own requirements. Users can pick sizes from small (font size 8), default (font size 12), and large (font size 16). By default, the general size of the fonts in the program will be sized 12.
Additional features		
10	Suggests to play similar algorithms	When users view a certain algorithm, the program also suggests an algorithm alike with the currently viewed one. This is to enhance better learning experience for users to seek out on similar problems
11	Appendix that shows further writeup of the algorithms available in the program	This shows the full writeup of the description shown during the animation, and additional information in regards with the algorithm.

2.1.2 Non-functional requirements

On the other hand, the non-functional requirements, refer to the set of requirements that describes how the system works in general, in terms of its operations. Table 2.3 below shows the set of non-functional requirements the system is expected to satisfy, along with its individual descriptions.

Table 2.3: Non-functional requirements of the software

No.	Requirements	Description
Graphical interface		
1	The images for the animation is to be scalable depending on the size of the user's input	The physical size of the animation highly depends on the input size given by either the user or the random generator. Due to this, the program needs to carefully scale the animation when it is either too small or too big for the screen. It needs to ensure that the user can easily see the images and fonts of the animation, whether the input size is small or large.
2	Tables included in the animation demonstration are to be scrollable when it gets larger than a specified size given	Some algorithms require a table, especially the dynamic programming types. The table varies in size depending on the size of input for the algorithm. If the table width and length gets bigger than a specific size given, instead of exceeding the size, the program is to add a scrollable feature for the table.
3	The program is to be clear and easy enough for users to comprehend its design	The colour scheme of the program is to have a calming, non-blaring proposition. The images and fonts along with it needs to be shown clearly, and easily relatable for the general public.
Settings		
4	Saves the settings provided by user	The program is to save the changes made by user under settings. This means that when the user opens the program again, the changed settings will still be in placed.

2.2 UML case diagram

The use case diagram below on figure 2.1 is the representation of what the user can do to interact with the system represented in use cases. It is basically shows the relationship between the user and the *Animated Algorithm Program*.

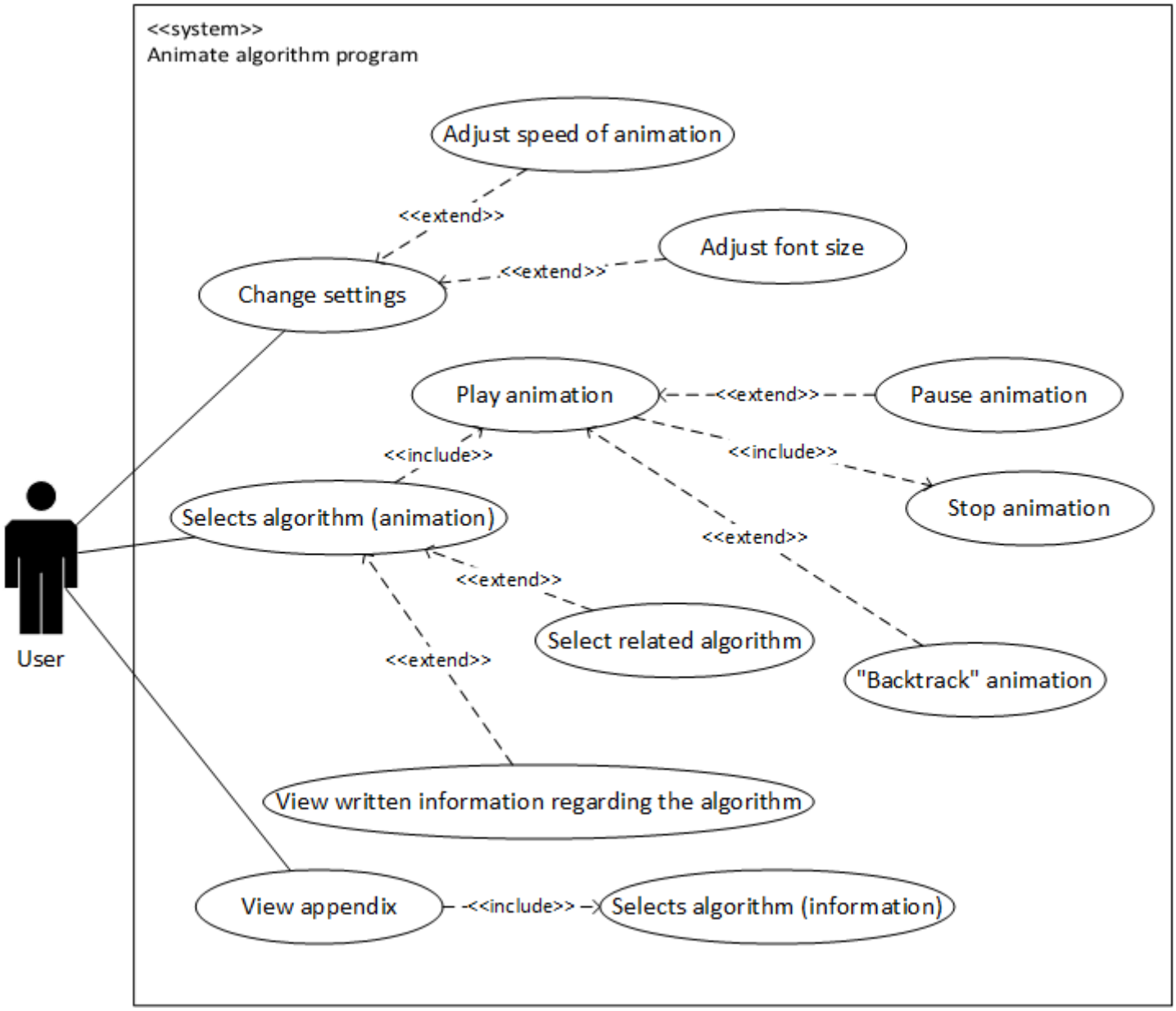

Figure 2.1: The system use case diagram

According to the use case diagram shown on figure 2.1, the user can change the settings of some of the features of the program, such as the speed of the animation, and its font size. This is to ensure that the user will be able to work comfortably within the environment, when the program allows the them to change these features of the program. The user could always revert back to the system's default settings if there is a need to.

Other than the settings, the animation is another feature that would be included within the program. By selecting an algorithm from the list of algorithms, the program will lead the user to the page where the animation of the algorithm resides in. From here, the user can manipulate the animation, by pressing controls such as play, pause, stop and "backtrack". The user can also access other algorithms that are *related* to the current one, if they ever wish to

do so. If the user clicks on the algorithm from the list of related algorithms, the program then goes to the animation page of the algorithm that was selected by the user.

Other than the settings and the animated feature, there will be an appendix within the program, where the users can view more information about the algorithm. The appendix however will only list the algorithms that are available in the program as animations. Within this feature, the appendix will mainly include a more detailed explanation about the algorithm that is not mentioned in the animation page. The information that can be found in the appendix is expected to contain predominantly texts and images of the algorithm.

2.3 System flowcharts

In this section, I have included the flowcharts of the program, to display the flow of the system in general, and how decisions controls its following events.

2.3.1 Main flow of the program

The flowchart below on figure 2.2 shows the overall flow of the program itself. The three main modules which makes up the program, i.e. change settings, view the animation of algorithms, and the appendix, are grouped in its respective subprocesses. Each subprocess will be described more in detail in sections 2.3.2, for the animation module, and 2.3.3 for the appendix module.

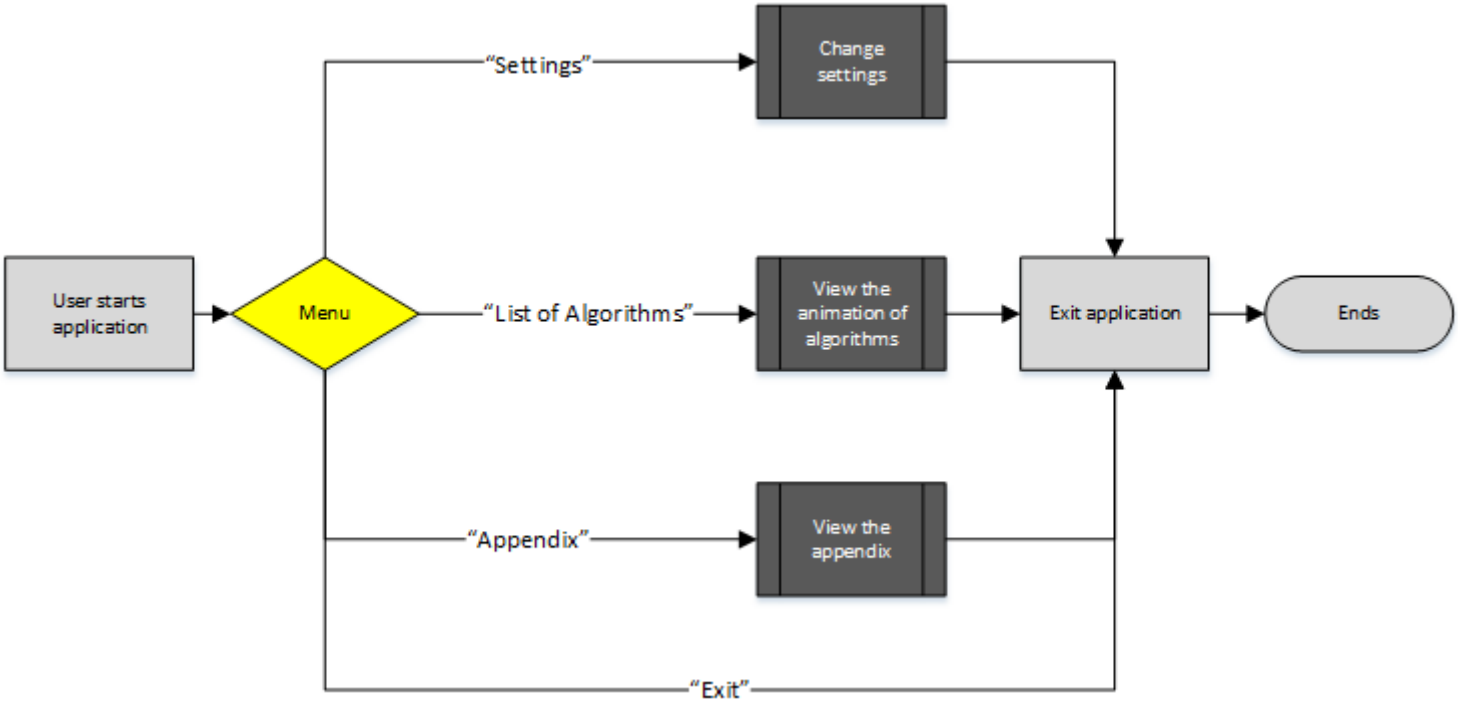

Figure 2.2: The flowchart of the whole system.

On the main flow chart below, the user first starts the application, which then brings them to the main menu. From here, the user can select up to three features they wish to use, which are *Settings*, *List of Algorithms*, and *Appendix*. The user can also choose to exit from the main menu, by selecting *Exit* which then closes the whole application.

2.3.2 Flow of the animation module

The animation module is admittedly the main feature of the program which will be heavily concentrated during the course of the implementation of the project. The user first selects the *List of Algorithms* button, that leads to the animation module. From here, a list of the algorithms are classified between 3 main paradigms and a sorting algorithm into different sections for user feasibility purpose. When the user selects an algorithm they wish to learn more about, the program would lead them to a page where the animation is, and a brief description about the algorithm shown in the *Information* section placed below the animation.

The program will first prompt the user either to enter their own specific input, or the generate a random value instead. When the user decides to add their own input, there will be a specific limit assigned to the algorithm. If the input exceeds the limited amount, the program

will throw an error message to inform user that the input was unacceptable, and requests the user to add an input that does not exceed the assigned limit. On the other hand, if the user selects the *Generate random value* button instead, the program would then generate a random value within the limited amount assigned, and input those value into the animation.

Once an input has been either retrieved, the user then will able to play the animation by pressing the play button. Whilst the animation is at its *playing state*, the user can control the animation by either *pause*, *backtrack*, or *stop* the animation. The state of the animation depends on the type of controls that have been selected by the user. To view the description of each control available in the animation page, below on table 2.4 that shows the outcome of the animation's state when a particular control button has been selected by the user.

Table 2.4: The list of animation controls.

Control	**Description**
Play	This button simply initiates the animation. This button is only available for use when the animation is either at its initial stage, *paused*, or *stopped*.
Pause	When a paused button is activated whilst the animation is playing, the animation stops temporarily. The stopped time will be saved, and will continue from that time if whenever the user chooses to play the animation. User can only pause the animation when the animation is being played.
Backtrack	This is a unique feature that comes in with the program. As the animation is animated through the use of *iterations*, these iteration values will be counted and stored programmatically. When a user clicks this button, the iteration counter, i, will be brought back to the previous iteration, which is i - *1*. Once it goes back to its previous iteration, the animation will be brought to its *paused* state. From here, the user can press *play*, which will initiate the animation from that particular state.
Stops	A stopped button will completely halt the process of the animation. Its final playing state will be discarded once a stop button is selected.

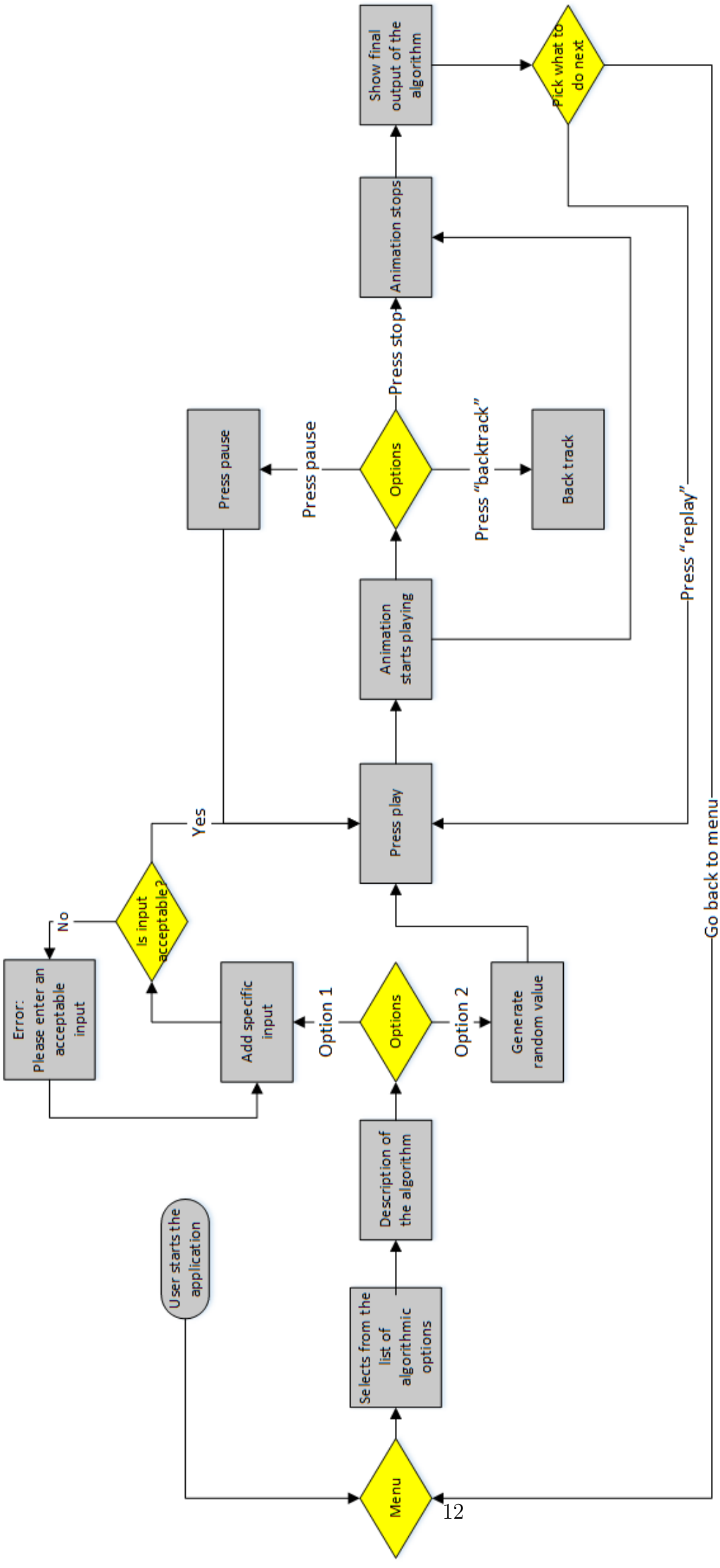

Figure 2.3: The flowchart of the animation module.

2.3.3 Flow of the appendix module

Finally, the last module would be the appendix module, which will contain the supplementary material in regards to the algorithms that are used in this program. This basically lists all the algorithms that are used, and users can select any algorithm within that list to view more information about the algorithm.

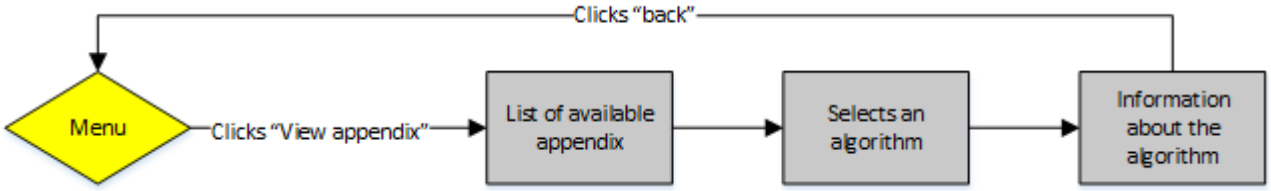

Figure 2.4: The flowchart of the appendix module.

The flowchart on figure 2.4 refers to the sequence of events that are involved within the module. First, from the main menu, as when the user selects *View appendix*, the program then brings the user to the list of all the available appendix found in the program. The user then could select the algorithm they wish to find out more about, by clicking one from the list. This will then lead the user to the page that predominantly presents the detailed information in regards to the algorithm in question.

2.4 Graphical User Interface design of the system

2.4.1 Program start page

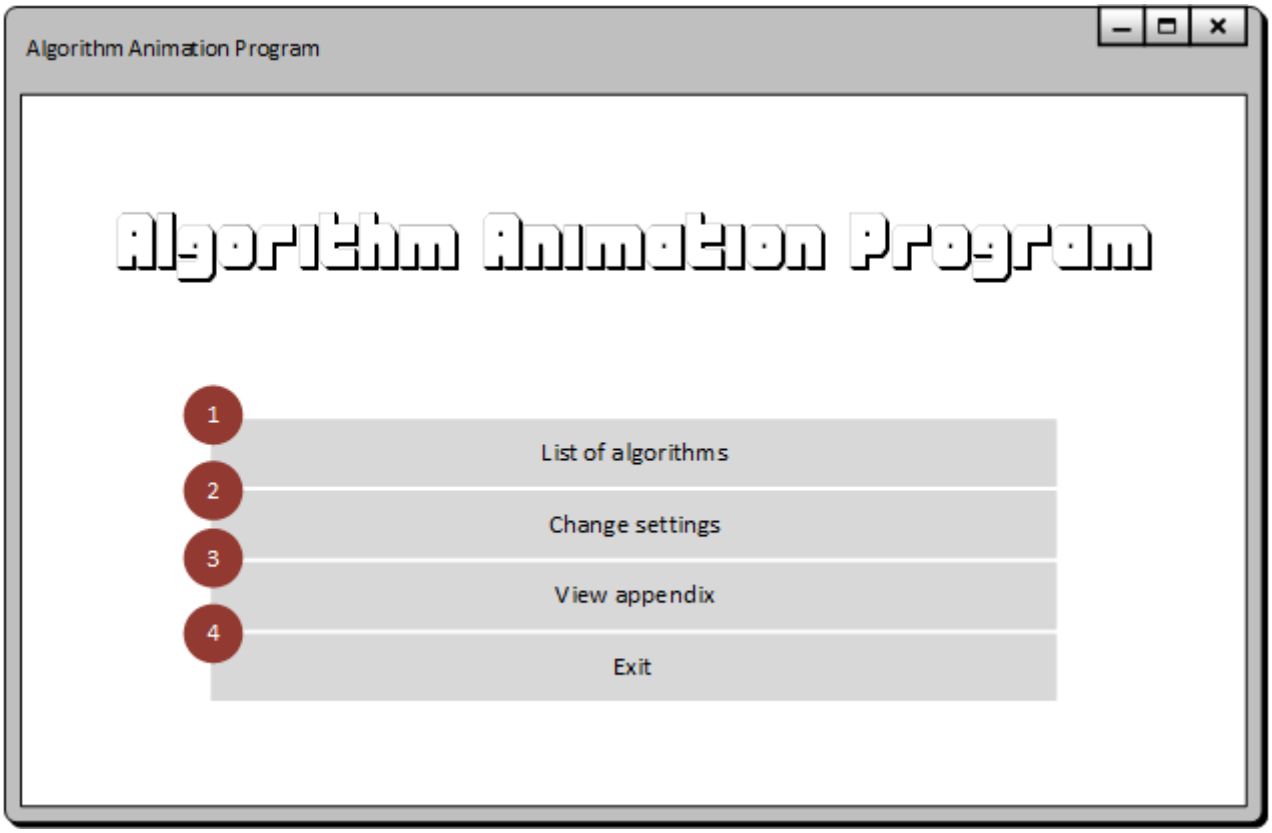

Figure 2.5: The start page of the program

As the program first initiates, the UI design shown in figure 2.5 will be the start page of the application. The start page displays the main menu of the application.

1. The *List of Algorithm* button leads the user to the list of algorithms. In this list, user can select whatever algorithm they wish to learn.
2. The *Change settings* button on the other hand, leads the user to a settings page.
3. The *View appendix* page brings the user to the main appendix list, which lists all the algorithms that is shown within the program.
4. Finally, the *Exit* button would close the whole application, if the user selects it.

2.4.2 Settings page

Figure 2.6: The settings page.

On the *Settings* page, users can adjust several features of the program, such as the font size, and the animation speed. The reason for having a settings page is to ensure that the users are working in an environment that they are most comfortable in.

1. One of the features the user can change is the font size. In order to change the font size, the user is to use the slider below. From the leftmost part of the slider is the smallest size of the font, which is size 9pt. The default size on the other hand is 12pt, followed by the largest possible size is 18pt.

2. Secondly, the user can also change the animation speed. Initially, the animation will be running on a default speed of . However, if the speed is either too fast or slow for the user, the user could always adjust the speed by sliding to the leftmost bit of the slider for slower speed, and rightmost for a faster speed.

find the specific speed of the animation!

2.4.3 List of algorithms page

Figure 2.7: The page that shows the list of algorithms available for animation.

In this page basically shows the list of all the algorithms that are available in animation. These algorithms are also to be classified between the three main algorithmic paradigms, which are *greedy method*, *divide and conquer*, and *dynamic programming* approaches. Also, another classification would be the sorting algorithms will be included in the list as well.

1. Shows the list of all the algorithms available. These algorithms are also classified according to its respective algorithmic paradigms. Users can click any of those algorithms displayed in that list if they wish to learn more about them.

2.4.4 The page of the animation of the algorithm selected

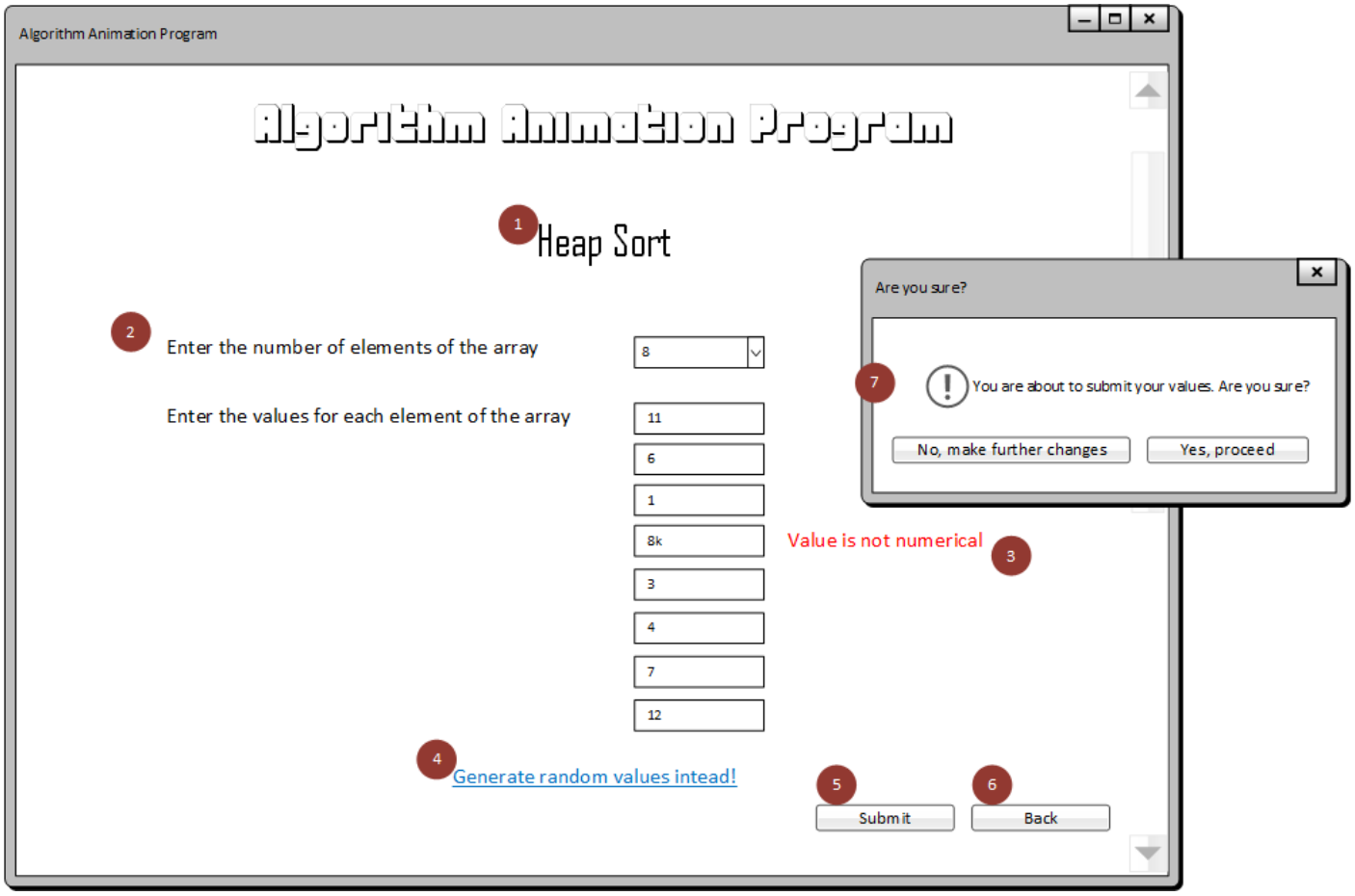

Figure 2.8: The page that shows requests the input from the user before starting the animation of the algorithm.

1. The title of the algorithm in question.
2. The form page of the list of input that is required for the heap sort animation.
3. As all fields are equipped with validation, it will throw an error message in regards to the field if it happens to receive the wrong input. For this case, the user fails to insert a numerical value.
4. Another option instead of adding the user's own input is to generate random values for the animation.
5. This button submits the values and proceeds to the next page, shown in figure 2.9.
6. This button simply brings the user back to the list of algorithms.
7. This dialog will be prompted once the user clicks the *Submit* button. This is to ensure that the user is happy with the input they have given. From here, if they click *No, make further changes*, the dialog will close, and the page remains the same. If the user clicks *Yes, proceed*, the program will proceed to the next page, which is shown in figure 2.9.

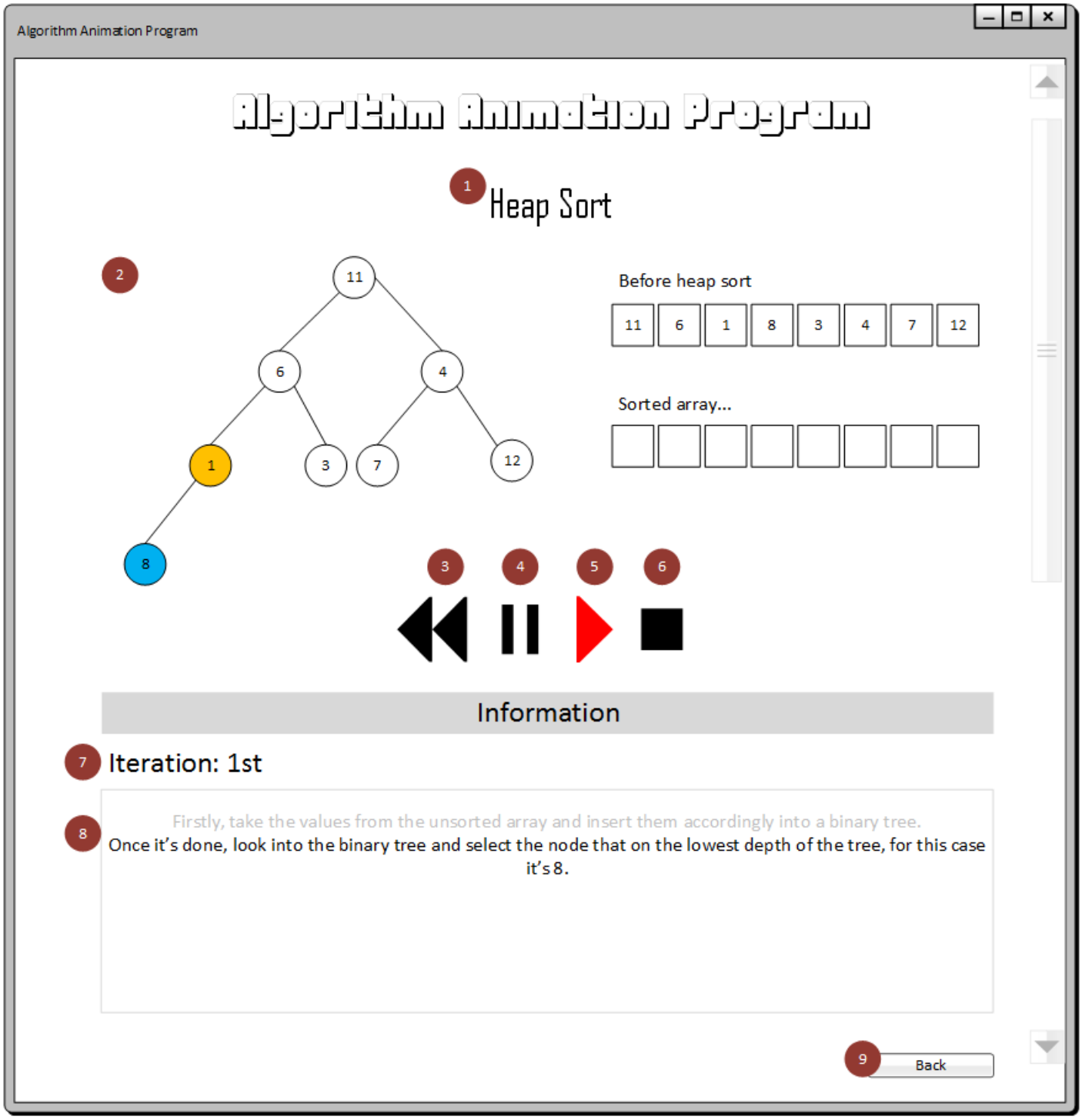

Figure 2.9: The page that shows the animation of the algorithm.

1. The title of the algorithm in question.
2. The section of the page where the animation of the algorithm is carried out.
3. The backtrack button.
4. The pause button.
5. The play button. Turns red when the animation is at the *playing state*. This applies to other control buttons as well.
6. The stop button.

7. The number of iteration the animation is currently at. Every time the animation finishes its *main loop*, the iteration counter is added, and it will be displayed here.

8. The area where a written information regarding the animation is displayed. Every time an animation displays something new, a new text block is displayed here, along with the numbers (variables) involved the animation. Once the animation has moved on to the next step, the text mentioned will be greyed out, as the new text will be the one that is emphasized.

9. The button simply brings the user back to the previous page which would be the main list of algorithms. Whatever that is played in the animation will be discarded.

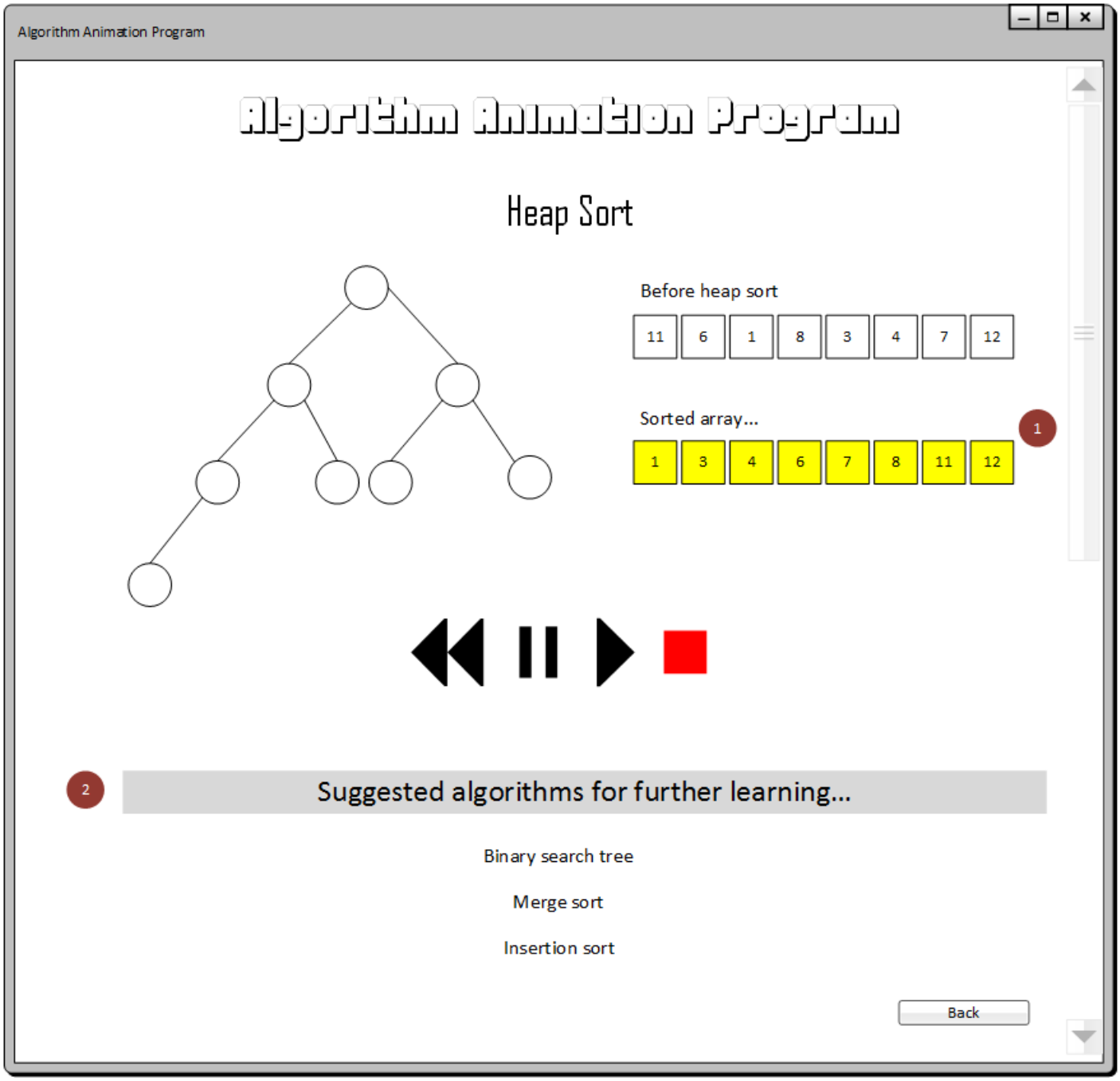

Figure 2.10: The page's layout when the animation has finished playing.

1. The animation stops right when the array has been fully sorted (only applies in sorting algorithms), or in general, has satisfied its respective goal(s). For this example, the sorted array is highlighted in yellow to show that the values in the array has been fully sorted.

2. Secondly, once the animation has ended, it also suggests the user on algorithms that are closely related to the current one for a comprehensive learning experience.

2.4.5 The appendix page

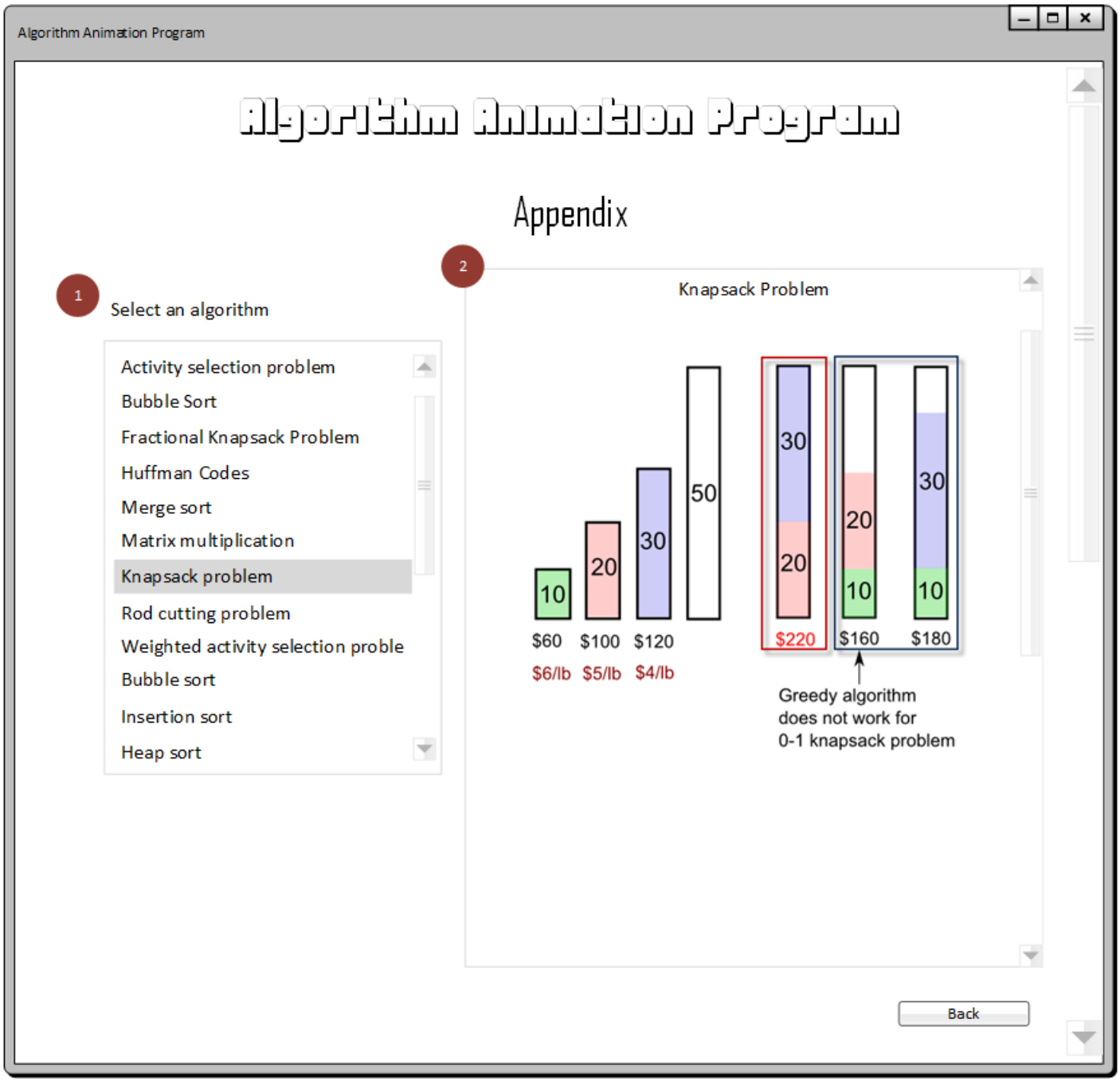

Figure 2.11: The appendix page which lists the algorithms available in the program, which then shows further information regarding the algorithm.

1. This section lists all the algorithms available in the program.
2. This section on the other hand displays whatever information about the algorithm selected from the list in no. 1.

2.5 UML class diagram

2.6 UML Sequence Diagram

Next up, is the sequence diagrams. These diagrams are primarily used to show the interactions between its objects in a sequential order, and the interactions that occur afterwards. The purpose of this diagram is to show in greater detail, compared to the class diagram shown in the previous section (section **??**), of what objects that are involved during a particular event. Do take note that these diagrams will be separated between the three main features of the program, the *animation*, *settings*, and finally *appendix*.

2.6.1 The animation feature

The sequence diagram for the animation feature shown in figure 2.12, reveals the interaction between the user and the different objects that are expected to be involved within this feature. First and foremost, as the user initiates the program, the program will display the main menu by executing the *displayMenu()* method. From there, the user selects (for this case) the *List of algorithms* button, which leads the user to the main list of algorithms.

The user then can pick any algorithm they wish to learn from this list. For example, the uses decides to learn how the merge sort algorithm works. So the user clicks the button called *Merge sort algorithm*, which then leads the user to the next page, the input request page, or known as *AlgorithmAnimationInputRequest* according to the sequence diagram in figure 2.12.

Once the values have been inserted, the program proceeds to the animation page by calling out the *displayAnimation()* method, along with the *play()* method that plays the animation. Once the animation is playing, the user can manipulate the animation by using the control buttons such as the *play*, *pause*, *stop*, and *backtrack* button. Each of these controls has its own call of method, and will be called accordingly depending on which control button is pressed.

If any control button is pressed, the program then calls out the *displayAnimationRealTime()* method. This method predominantly used to run the animations, such as keeping track of the iteration(s) of where the animation is currently at, and also tracks where the timing of the animation is currently at when a control button is clicked. Of course, the latter does not apply to the *stop* button, as the animation will be told to immediately reach to the end once the animation has stopped.

Once the animation has stopped, either by the user or the animation simply reaches to its own end, the program will then give the user an option of viewing other algorithms that are similar to the one in question, by calling the *getSuggestedAlgorithms()* method. The user can either click one of the suggested algorithms, or choose to go back, which brings them to the main algorithm list.

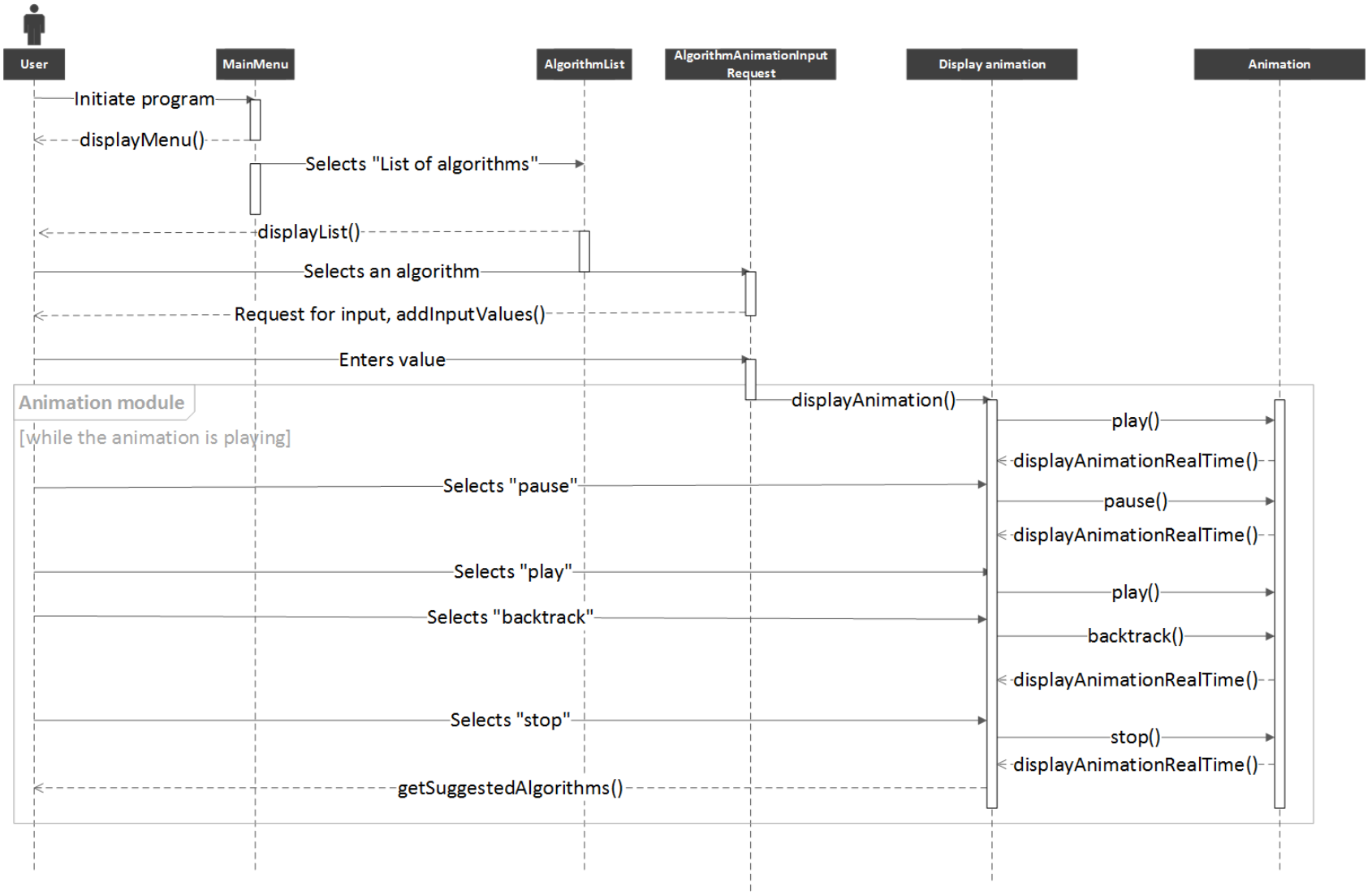

Figure 2.12: The sequence diagram of when the user uses the animation feature.

2.6.2 The settings feature

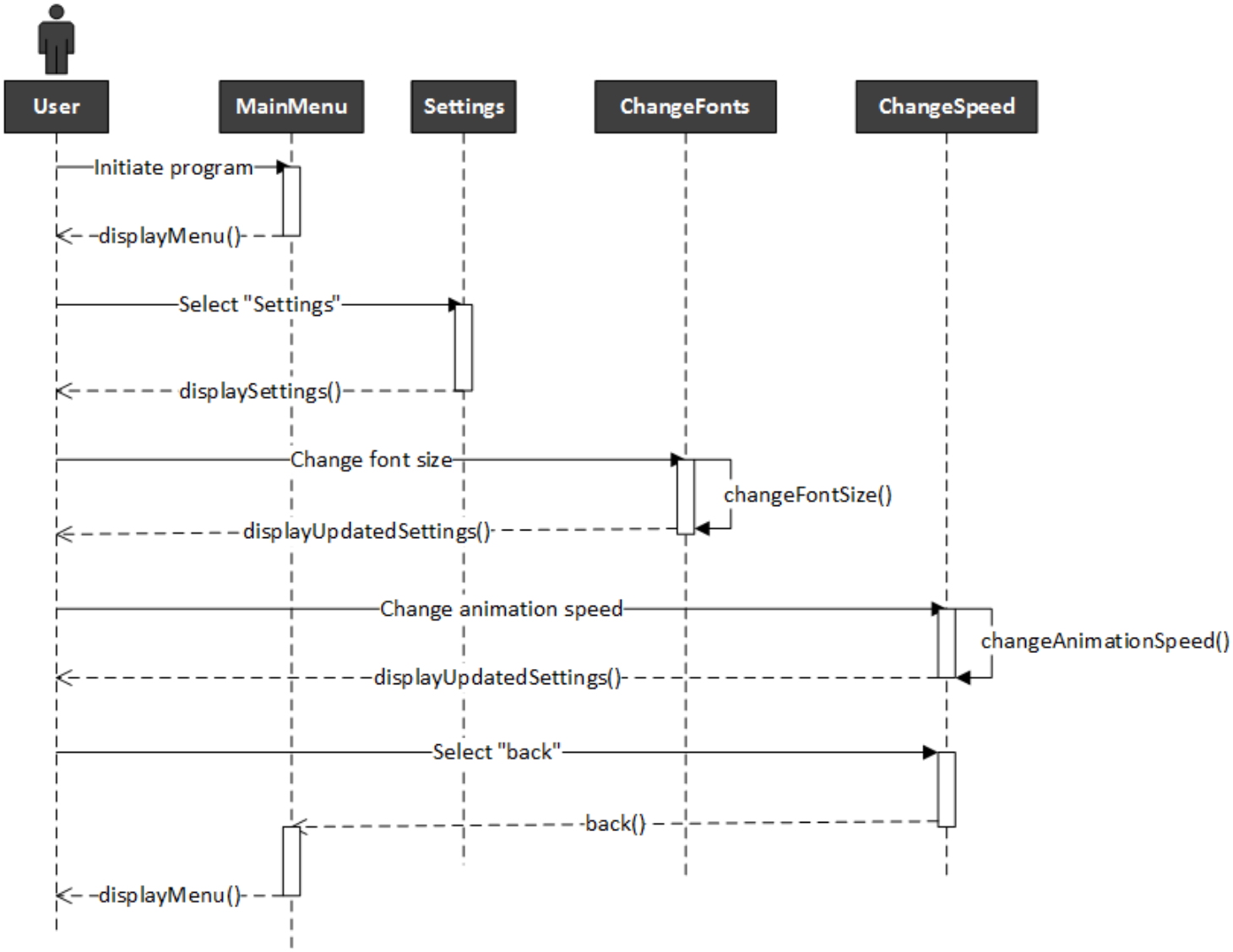

Figure 2.13: The sequence diagram of when the user uses the settings feature.

Another feature that is available in the program would be the settings feature. Based on the sequence diagram in figure 2.13, as when the user opens the program, the program will call the *displayMenu()* method to display the main menu onto the client. The user then selects *Settings* button, which leads them to the settings page, which would be called out by the *displaySettings()* method.

Once the user is in the settings page, the user can change features such as the font size and the animation speed. If the user decides to change the font size, the *changeFontSize()* method is being called out. When the method is called, it passes the parameter of the font size chosen by the user, and uses it to make necessary changes to the font size. Once that is done, the program then calls out the *displayUpdatedSettings()* to change the outlook of the program to whatever that is changed. The same applies to when the speed of animation is changed, however this time, the *changeAnimationSpeed()* method is called out instead.

Once the user is done with changing the settings, they could then press the *Back* button, which leads them back to the main menu. To see the graphical interface design of the settings page, do refer to figure 2.6, on page 15.

2.6.3 The appendix feature

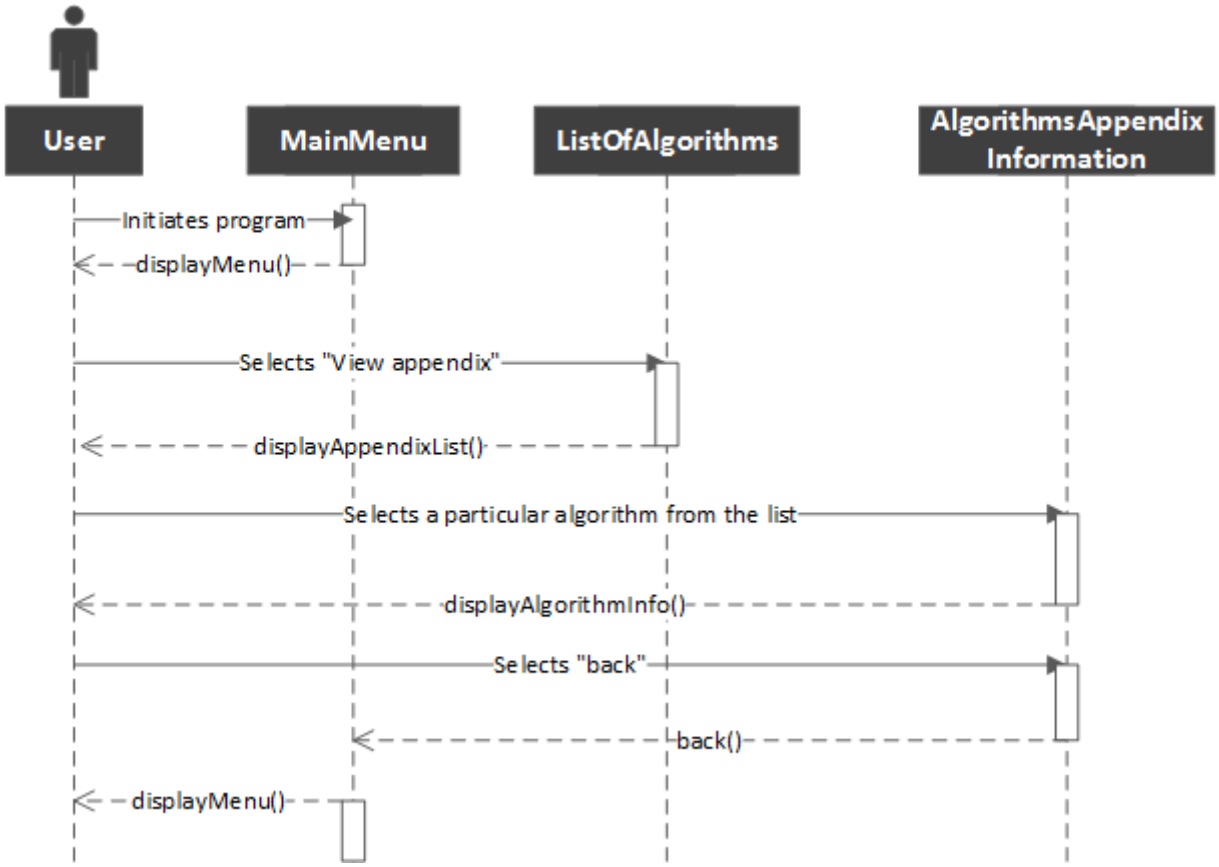

Figure 2.14: The sequence diagram of when the user uses the appendix feature.

Last but not least, the appendix would be the other feature that would be available in the *Algorithm Animation* program. From the main menu, the user clicks the *View appendix* button, which calls out the *displayAppendixList()* method, and leads the user to the appendix page. Once the appendix page is displayed, the user could select the algorithm they wish to find out more about, within the list presented on the page. Once the user selects one, the program calls out the *displayAlgorithmInfo()* with the name of the algorithm as the passing parameter, which displays the information available for the algorithm.

When the user is done, the *Back* button can be selected, which leads them back to the main menu. Like before, if you wish to refer to the graphical user interface design of the appendix page, refer to figure 2.11 in page 21.

2.7 Algorithm animation designs

In this section, I will present you the designs of the animations that will be included in the *Algorithm Animation* program. This section is to propose an idea how the animation will look and act like, and to give a guidance on how to develop them during the implementation stage. Alongside with the design of the animation, I will also include a brief description about the algorithm, and whatever is happening within the animation.

Note that I have not included the control buttons in some of these designs. This section is mainly to emphasize the design of the animation of the algorithm in question. If you would like to refer to the design of the actual graphical interface within the animation page, do have a look at section 2.4, on page 14 for more details.

2.7.1 Fractional Knapsack Problem

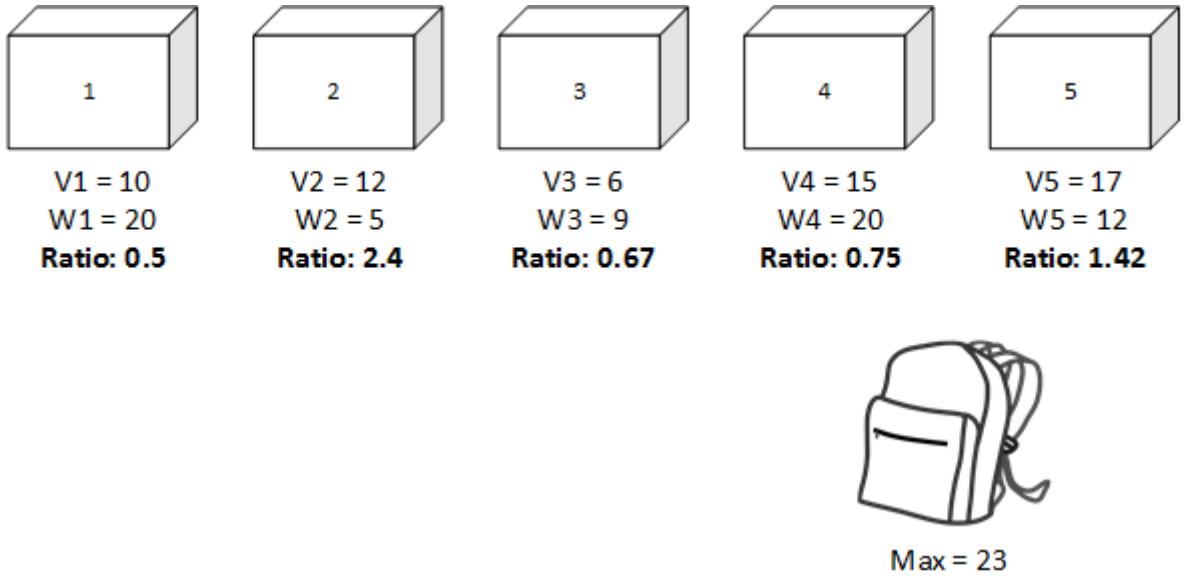

Figure 2.15: The animation design of the FractionalKnapsackProblem. This shows the initial state of the animation.

The Fractional Knapsack Problem is one of the algorithms that uses a combinatorial optimisation strategy in which the goal is to fill the knapsack with fractional amounts of items until it reaches to its maximised value. This algorithm predominantly uses the greedy method approach in order to reach to its optimal solution. Within this section, I have included the design

of how the animation for this algorithm will turn out to be, along with its descriptions that describes each of the different processes that happens during the play time of the animation.

The graphical interface of the Fractional Knapsack Problem algorithm shown in figure 2.15 is the initial state of the animation. The animation consists of the items, which are represented in cubes, along with its values, i.e. weight and value of the item. Other than the items, there will also be an image of a rucksack that represents the knapsack. The knapsack has its max weight, which is the maximum amount of weight it can possible carry, and the current weight it is carrying now.

The next step of this animation shown in figure, is simply to follow the process of the animation. In this design, the animation is currently at the stage where it needs to calculate each of the items' value to weight ratio. The animation simply adds another value to each of those items called *Ratio*, that shows the results to the calculation. Next would be the part of the animation is when the items are re-arranged according to their value to weight ratios.

Once the items are finally arranged, it is now time to add the items into the knapsack. The animation will represented that by dragging the item in its current iteration next to the knapsack. An arrow will then appear next to the item to represent the act of adding the item into the knapsack. When the item in question is added into the knapsack, there will be a change on the *Current weight* value.

There is an exception to this animation, that the knapsack, can no longer contain the total weight of the item in question. So, the animation program will inform the user, under the information section, that it can't insert the whole of item in question. This proceeds the animation to add fractions of the item instead, and doing a simple calculation (ratio * amount of weight left in the bag) along it as well. You can clearly see this calculation clearly above the arrow.

As when the calculation is complete, and the knapsack's current weight has reached to its max weight, the animation then comes into a conclusion. The conclusion of this algorithm is that the knapsack is finally full, and the total value that the knapsack has achieved is displayed as well. Afterwards, the animation stops here.

2.7.2 Knapsack Problem

The Knapsack Problem is quite similar compared to the Fractional Knapsack Problem. However for this problem, you can only take the whole item and leave the rest if there is no space left in the knapsack, instead of taking items fractionally until the knapsack is full. Despite a minor change in the constraints of this problem, the approach to the knapsack problem is very different to the fractional one. Instead of utilising the greedy method like we would do for the fractional, this requires us to use the dynamic programming approach instead. Because of this, the animation needs to add another feature, and this will basically be applied to all algorithms that uses the dynamic programming approach as well: tables.

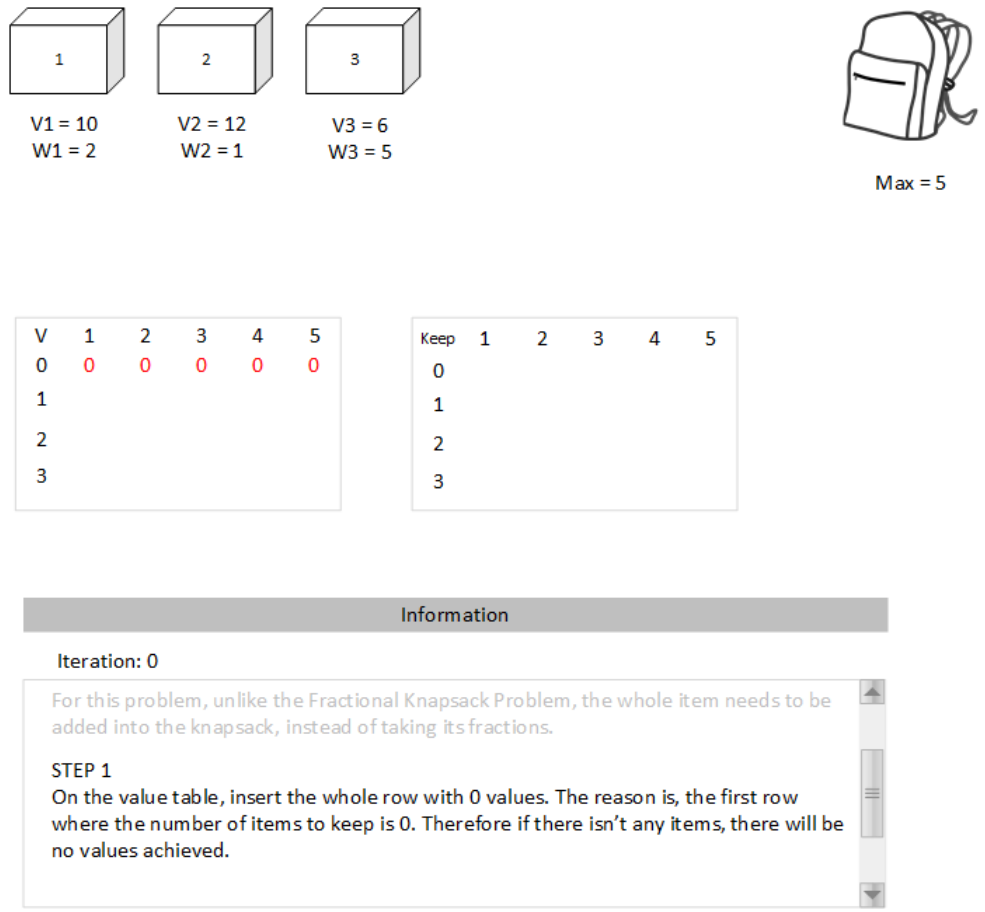

Figure 2.16: The initial state of the animation design of the Knapsack Problem. It shows that it is describing what the tables are.

As when the animation of the knapsack problem starts (refer to figure 2.16), few description which describes about the tables will be displayed during the animation's initial play time. Those descriptions in the yellow coloured callouts are placed around the table to describe its purpose, in order for the user to understand what they are before the actual algorithm starts. These texts will remain here for about a 30 seconds, before proceeding to the next state.

Included in this animation, other than the ones mentioned before, are the cube figures which represent the items. Along with them are its attributes, which are their respective value they are worth, and their weight. Next to it is an image of the knapsack. Show below it is its maximum capacity, and the current weight it is holding. For its initial state, the current value is always 0.

Within the *Information* section, there will also be a brief description of what the algorithm is, and how is it different from the Fractional Knapsack algorithm.

After the explanation phase has completed, the algorithm of the knapsack problem begins. The algorithm begins by applying the sum of values obtained on the first row of the *value table*. Since on the left side of the table is 0, meaning that there are no items selected, all the values are returned as 0 for the first row. The texts that are inserted into the table are turned red, simply to let the users know where to look in the animation. The values are added into the *keep table* as well after adding those values into the *values table* previously.

As the animation proceeds to the second row, where the animation loops by doing the same thing previously on the first row. It refers to the top column, which is the weight of the knapsack to carry, which is highlighted with a green circle. Followed by the spot where the animation needs to compute the value for, which is highlighted with a red circle.

Once filling up the *value* and *keep* table is completed, the algorithm looks through the keep table from the most bottom right hand side, to find out which item to keep. If the value is 1, it means that the item along that row is kept, and the total weight of the knapsack will be subtracted from the item. Do this along the column which is the maximum possible amount of capacity of the knapsack, in this case 5, until either the knapsack is full, or can no longer take in any more items. Finally once it has done that, show the total value (most optimum value you can achieve) that is taken away from the knapsack.

```
// Input:
// Values (stored in array v)
// Weights (stored in array w)
// Number of distinct items (n)
// Knapsack capacity (W)

for j from 0 to W do:
     m[0, j] := 0

 for i from 1 to n do:
     for j from 0 to W do:
          if w[i] <= j then:
               m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
          else:
               m[i, j] := m[i-1, j]
```

codes/knapsackProblem.java

describe the psuedo code

2.7.3 Activity Selection Problem

In the Activity Selection Problem, you are given a set of activities, where each have its respective starting and finishing time. Within this set of activities, the goal is to find the most number of activities that can run without interrupting each other. This algorithm uses the greedy method approach in order to find the optimum set of activities.

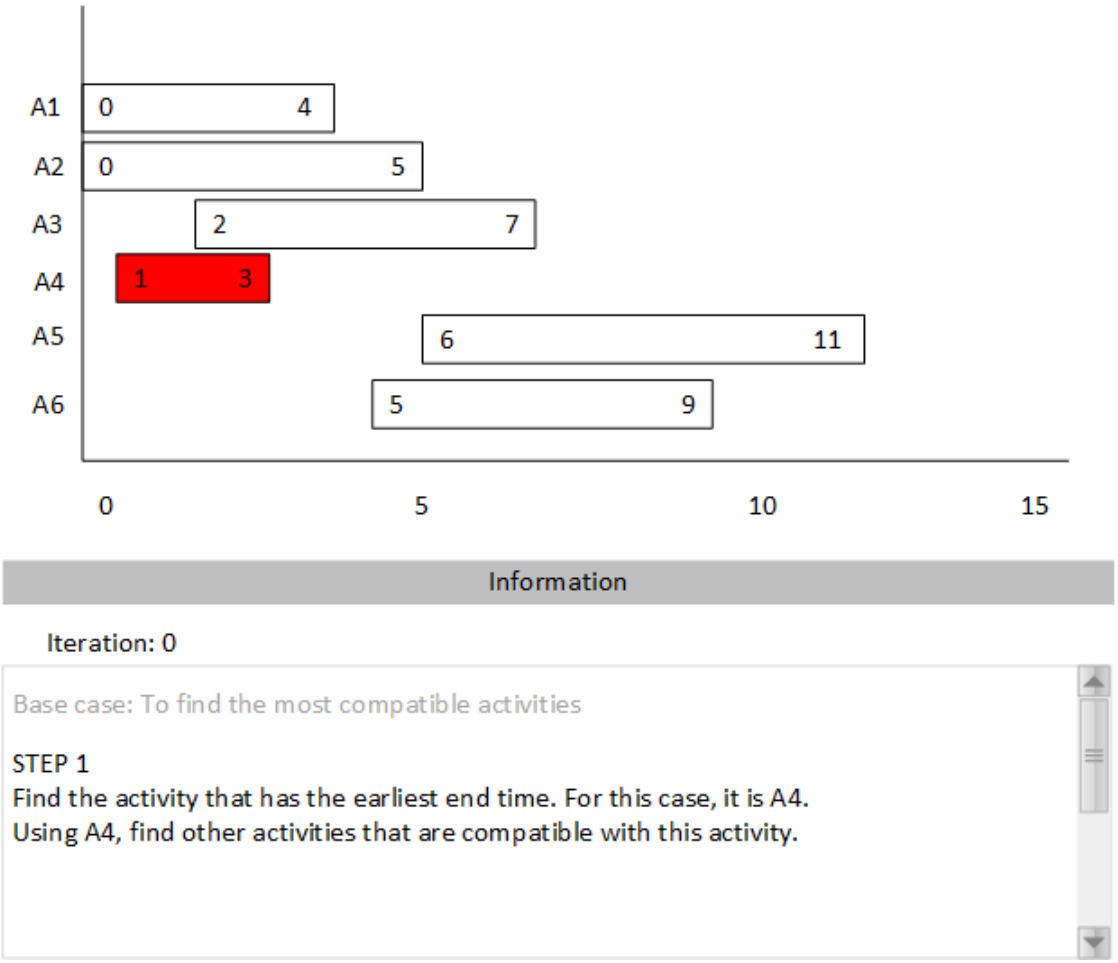

Figure 2.17: The animation design of the Activity Selection Problem.

Shown in figure 2.17, is the initial stage of the activity selection animation during its starting phase. During this phase, the animation starts off by explaining parts of the diagram, using the bright yellow callouts. Along with it, there is a further explanation about the objective of the algorithm within the information section below.

Once the explanation phase is done, the animation carries on by initiating the activity selection problem algorithm. First, it selects the activity with the earliest end time, and highlights it red. Once the activity has been chosen, it goes on by finding the other activities within the list that is incompatible with the chosen activity. These incompatible activities are then removed from the list of activities, and are greyed out in the animation to show that they are removed. Once there aren't any more activities left in the list of activities, the optimal solution to the problem has been finalised. This is when the animation ends.

```
1  Greedy-Iterative-Activity-Selector(A, s, f):

3       Sort A by finish times stored in f'

5       S = {A[1]}
        k = 1
7

        n = A.length
9

      for i = 2 to n:
11       if s[i]      f[k]:
             S = S U {A[i]}
13           k = i

15    return S
```

codes/activitySelectionProblem.java

describe the psuedo code

2.7.4 Merge sort

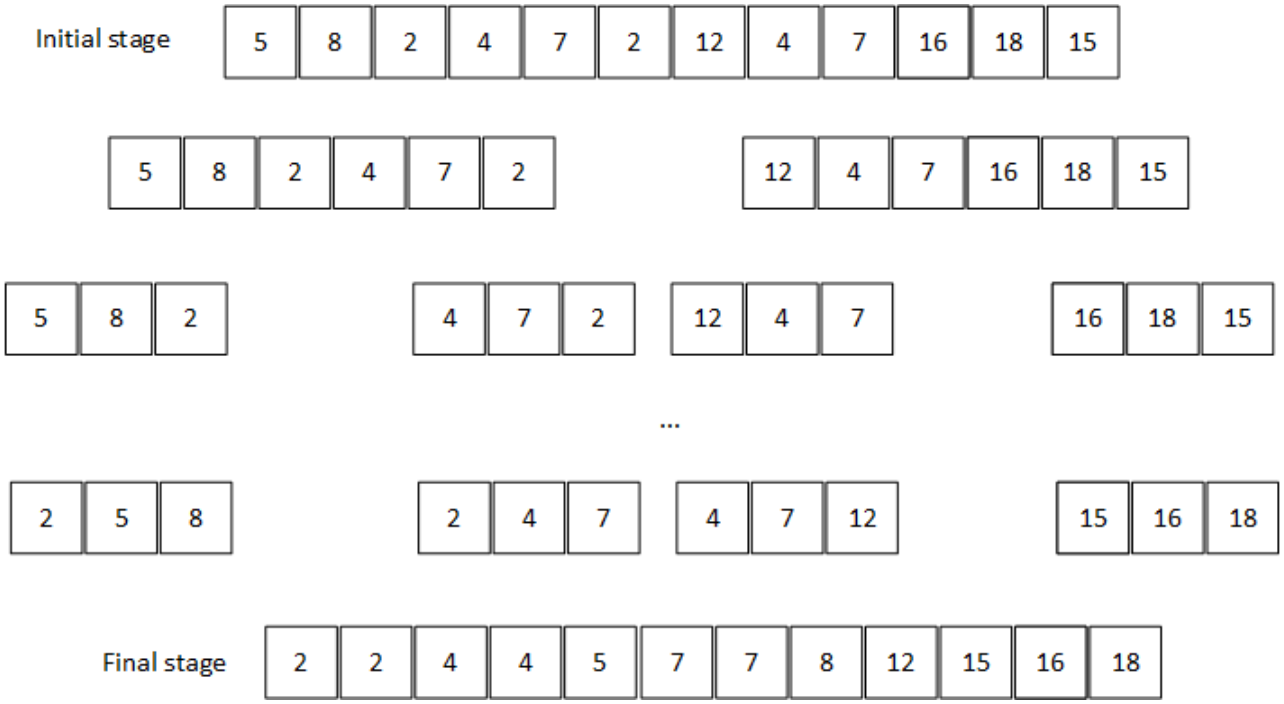

Figure 2.18: Merge sort animation design.

The merge sort algorithm is basically one of the sorting algorithms that uses the divide and conquer method to sort the contents of the array of values. The animation for the merge sort algorithm will emphasize mainly on the idea of a divide and conquer, which is one of the three algorithmic paradigms.

On the input page for the merge sort algorithm, will be the number of elements in array, followed by the values for each element of the array. The animation for this algorithm will be made simple, as you can see in figure 2.18. First, is to have squares that represents each element of the array. These squares will initially be set next to the other squares with values clearly shown on them.

As when the animation starts, the animation will clearly show the division it makes to divide the array into 2 parts, each side with an equal amount of elements to the other side (unless if the total number of elements is an odd number, then one side will have one more element than the other). The animation will keep on doing this until each of the elements divided will be left to one each. After which, the animation will proceed in taking the first element from the unsorted list, sorts and drags it down to another tier, and do the same for the rest until all the elements are sorted and merged as a whole. The animation then concludes by showing that it is finally sorted, and ends here.

```
/* array A[] has the items to sort; array B[] is a work array */
void BottomUpMergeSort(A[], B[], n)
{
  /* Each 1-element run in A is already "sorted". */
  /* Make successively longer sorted runs of length 2, 4, 8, 16... until whole
   array is sorted. */
  for (width = 1; width < n; width = 2 * width)
    {
      /* Array A is full of runs of length width. */
      for (i = 0; i < n; i = i + 2 * width)
        {
          /* Merge two runs: A[i:i+width-1] and A[i+width:i+2*width-1] to B[] */
          /* or copy A[i:n-1] to B[] ( if(i+width >= n) ) */
          BottomUpMerge(A, i, min(i+width, n), min(i+2*width, n), B);
        }
      /* Now work array B is full of runs of length 2*width. */
      /* Copy array B to array A for next iteration. */
      /* A more efficient implementation would swap the roles of A and B */
      CopyArray(B, A, n);
      /* Now array A is full of runs of length 2*width. */
    }
}

void BottomUpMerge(A[], iLeft, iRight, iEnd, B[])
{
  i0 = iLeft;
  i1 = iRight;
  j;

  /* While there are elements in the left or right runs */
  for (j = iLeft; j < iEnd; j++)
    {
      /* If left run head exists and is <= existing right run head */
      if (i0 < iRight && (i1 >= iEnd || A[i0] <= A[i1]))
        {
          B[j] = A[i0];
          i0 = i0 + 1;
        }
      else
        {
          B[j] = A[i1];
          i1 = i1 + 1;
        }
    }
}

void CopyArray(B[], A[], n)
{
    for(i = 0; i < n; i++)
        A[i] = B[i];
}
```

codes/mergeSort.java

describe the psuedo code

2.7.5 Matrix multiplication

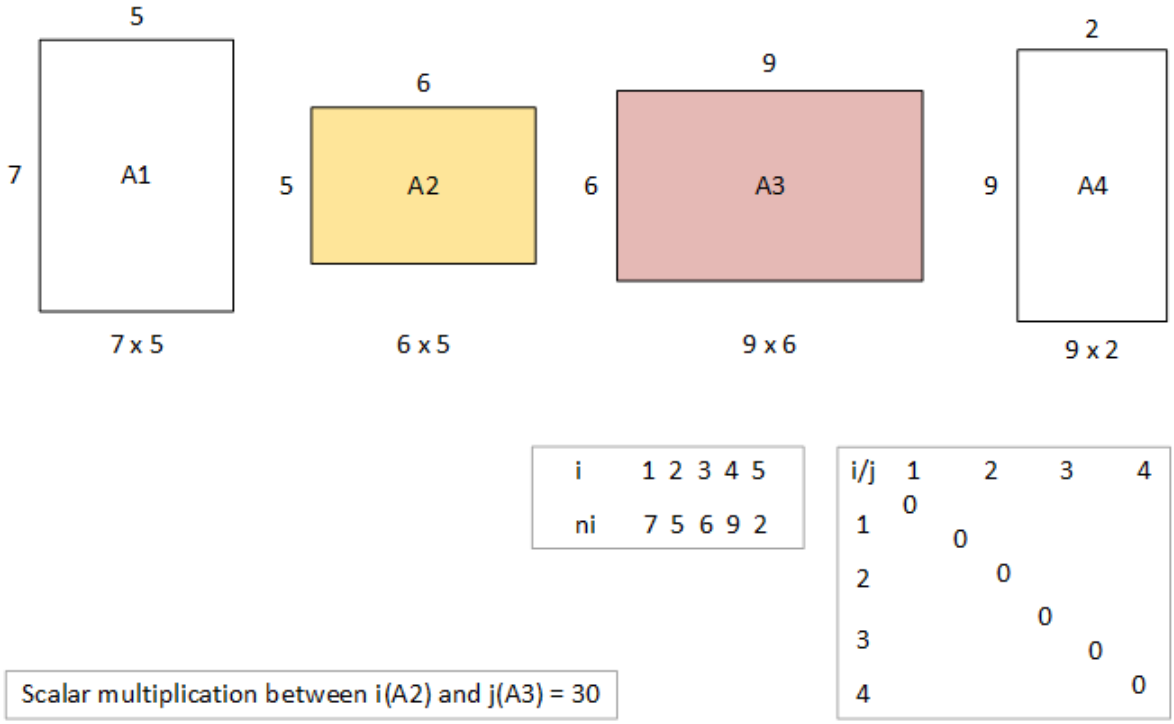

Figure 2.19: Merge sort animation design.

The matrix multiplication problem is classified under the *optimisation problem*, that will be available in animation. This problem simply shows the algorithm on finding the most efficient way to multiply two or more matrices together, and using dynamic programming in order to find the solution.

The input page for the animation will consist of the number of matrices to compute and the size of each of those matrices. There will be some constraint when it comes to selecting the size of the matrix however, is that, the next matrix will need to have one of its sides to be the same value as one of the sides the first matrix as well, which is a basic rule for two matrices to be multiplied with one another.

On the animation itself, it will present each matrix in a rectangular shaped figures, depending on the size that was given during the input stage. On top of each matrix will also be an identifier, which format is *Mi*, *i* being the number of the matrix in the list. Below each of the rectangles will show the size of the matrix, which format goes as *widthSize* * *lengthSize*, in order to depict the size of each rectangles.

During a part of the animation, the algorithm will pick the values for i and j, which are the values between the 1, the first matrix, and N, where it is the number of the last matrix in the list. Between these two values, there will also be k, which is going to be used to compute the most efficient *split* to be made, in order to achieve the minimal amount of scalar multiplications between the two lists of the matrices. In every iteration of the algorithm, the animation is to use k as a splitter, and between the two parts of the list, find the minimum amount of scalar multiplications needed. And once the minimum has been found, the algorithm will conclude its answers, and ends.

Since this problem uses dynamic programming, there is a need to add a table that goes with the animation as well. This table is what I call the *i and j* table, which is a two dimensional array that makes up the size of i and j. This table is used to store the result of the scalar

multiplications between the size of the matrices, and is used to find the most efficient solution for the problem.

```
// Program in C# to multiply two matrices using Rectangular arrays.
using System;
class MatrixMultiplication
{
  int[,] a;
  int[,] b;
  int[,] c;

  public void ReadMatrix()
  {
    Console.WriteLine("\n Size of Matrix 1:");
    Console.Write("\n Enter the number of rows in Matrix 1 :");
    int m=int.Parse(Console.ReadLine());
    Console.Write("\n Enter the number of columns in Matrix 1 :");
    int n=int.Parse(Console.ReadLine());
    a=new int[m,n];
    Console.WriteLine("\n Enter the elements of Matrix 1:");
    for(int i=0;i<a.GetLength(0);i++)
    {
      for(int j=0;j<a.GetLength(1);j++)
      {
        a[i,j]=int.Parse(Console.ReadLine());
      }
    }

    Console.WriteLine("\n Size of Matrix 2 :");
    Console.Write("\n Enter the number of rows in Matrix 2 :");
    m=int.Parse(Console.ReadLine());
    Console.Write("\n Enter the number of columns in Matrix 2 :");
    n=int.Parse(Console.ReadLine());
    b=new int[m,n];
    Console.WriteLine("\n Enter the elements of Matrix 2:");
    for(int i=0;i<b.GetLength(0);i++)
    {
      for(int j=0;j<b.GetLength(1);j++)
      {
        b[i,j]=int.Parse(Console.ReadLine());
      }
    }
  }

  public void PrintMatrix()
  {
    Console.WriteLine("\n Matrix 1:");
    for(int i=0;i<a.GetLength(0);i++)
    {
      for(int j=0;j<a.GetLength(1);j++)
      {
        Console.Write("\t"+a[i,j]);
      }
      Console.WriteLine();
    }
    Console.WriteLine("\n Matrix 2:");
    for(int i=0;i<b.GetLength(0);i++)
    {
      for(int j=0;j<b.GetLength(1);j++)
      {
        Console.Write("\t"+b[i,j]);
      }
      Console.WriteLine();
```

```
    }
    Console.WriteLine("\n Resultant Matrix after multiplying Matrix 1 & Matrix 2:");
    for(int i=0;i<c.GetLength(0);i++)
    {
      for(int j=0;j<c.GetLength(1);j++)
      {
        Console.Write("\t"+c[i,j]);
      }
      Console.WriteLine();
    }

  }
  public void MultiplyMatrix()
  {
    if(a.GetLength(1)==b.GetLength(0))
    {
      c=new int[a.GetLength(0),b.GetLength(1)];
      for(int i=0;i<c.GetLength(0);i++)
      {
        for(int j=0;j<c.GetLength(1);j++)
        {
          c[i,j]=0;
          for(int k=0;k<a.GetLength(1);k++) // OR k<b.GetLength(0)
          c[i,j]=c[i,j]+a[i,k]*b[k,j];
        }
      }
    }
    else
    {
      Console.WriteLine("\n Number of columns in Matrix1 is not equal to Number of rows in Matrix2.");
      Console.WriteLine("\n Therefore Multiplication of Matrix1 with Matrix2 is not possible");
      Environment.Exit(-1);
    }
  }
}
class Matrices
{
  public static void Main()
  {
    MatrixMultiplication MM=new MatrixMultiplication();
    MM.ReadMatrix();
    MM.MultiplyMatrix();
    MM.PrintMatrix();
  }
}
```

codes/matrixMultiplication.cs

describe the psuedo code

2.7.6 Rod cutting problem

Rod length = 5

Price/Length		0	1	2	3	4	5
£2	1	0	2	4	6	8	10
£5	2						
£7	3						
£8	4						

Length	1	2	3	4
Price	2	5	7	8

Figure 2.20: Merge sort animation design.

The rod cutting problem is another one of those algorithms that uses the dynamic programming strategy. The problem consists of a rod, which is a x amount of length. Along with it, there is a table of prices, selling for a certain amount for a certain amount of length. Using some sort of a strategy (for our case, an algorithm), find the best cuts of the rod of x length, in order to get the most value out of it.

In the input page for the rod cutting problem, would have the input for the length of the rod, and the prices for each length of the rod. When the input is inserted, the animation starts by showing the image of a rod, with the length assigned to it. Below it as well are two tables, one of them, the *Price/Length* table, is the table that is used to calculate the price earned for selling the rod at a certain length. This table will be filled along the way during the course of the animation. The other table is the *price table*, which lists the set of length and its prices.

The algorithm ends when the *Price/Length* table is fully filled, and the optimum cut is retrieved from the table. The animation will then conclude the optimum price for the particular problem before it terminates.

psuedo code

2.7.7 Bubble sort

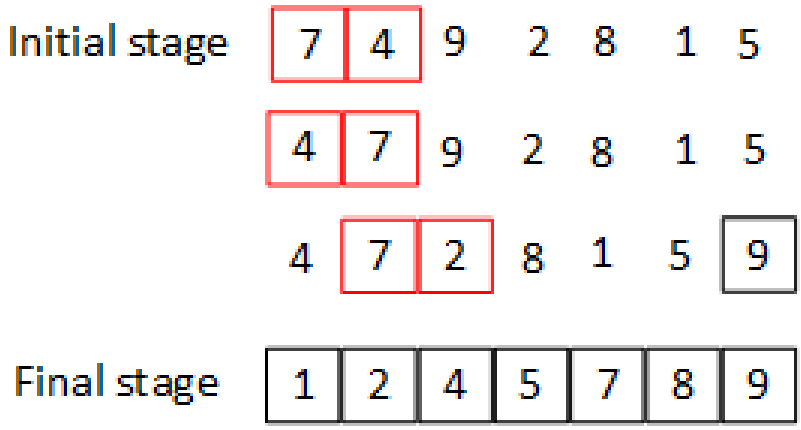

Figure 2.21: Merge sort animation design.

The bubble sort is one of the sorting algorithms that I will include in this program, classified under the *Sorting algorithms* section. It is a simple algorithm that compares each adjacent pair, and swaps with one another, left side is the smaller value, whilst the right side is the larger value. The algorithm keeps doing so until the whole list is fully sorted.

For the input page, the values that the program will take before starting the animation would be, the size of the array, and the values for each of the elements in the array.

The animation for the bubble sort algorithm will show the list of values, arranged accordingly to the input that was given by the user, or the random generator. These values are shown basically depict each of the element of the array, and the animation will show clearly that each of them will be swapped with the value next to itself clearly when it is its time to swap. Once the largest value in the list has moved all the way to the right side of the array, a black border will surround it, indicating that the position for this element has been finalised. The animation will come to an end, once all the elements have its own black border. The animation then concludes that the array of values have been sorted, and will stop itself then.

descripti
about
the
psuedo
code

```java
public static void BubbleSort( int [ ] num )
{
     int j;
     // set flag to true to begin first pass
     boolean flag = true;
     int temp;   //holding variable

     while ( flag )
     {
            //set flag to false awaiting a possible swap
            flag= false;
            for( j=0;  j < num.length -1;  j++ )
            {
                  // change to > for ascending sort
                   if ( num[ j ] < num[j+1] )
                   {
                          //swap elements
                           temp = num[ j ];
                           num[ j ] = num[ j+1 ];
                           num[ j+1 ] = temp;
                           //shows a swap occurred
                          flag = true;
                  }
            }
      }
}
```

codes/bubbleSort.java

2.7.8 Insertion sort

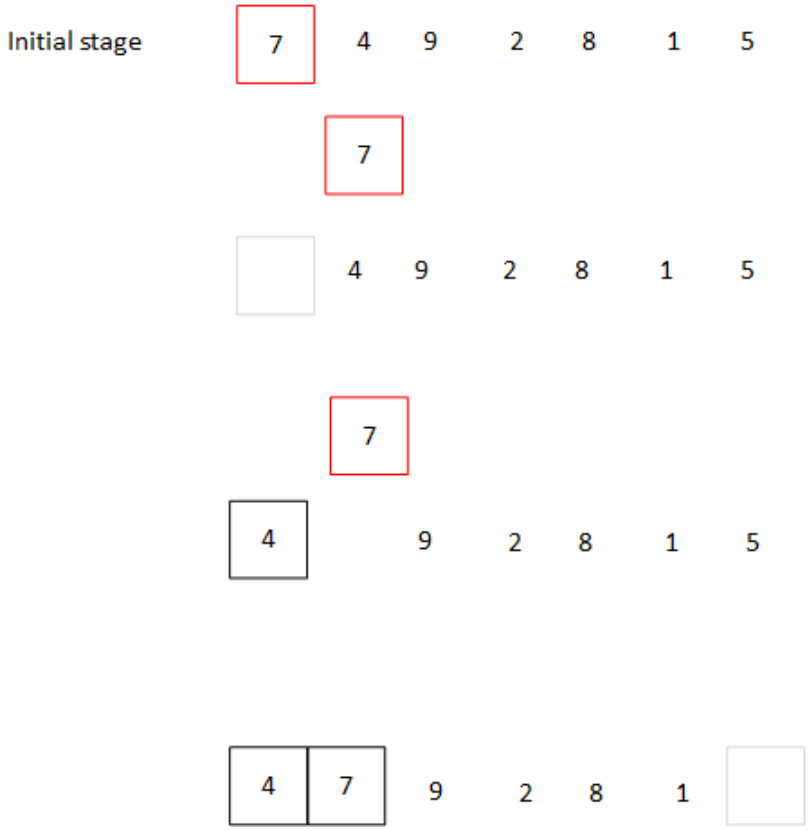

Figure 2.22: The insertion sort animation design.

The input page for the insertion sort will be very similar to the input page for the bubble sort. However, of course, the animation of the insertion sort will be different compared to it. At the initial stage, as seen in figure 2.22, the animation shows the values, where these values represents each element in an array. As the animation begins, the first value from the left will be highlighted with a red bordered square, and then"pulled out" from the main array. This value is then used to compare with the values from the left side, values which are in the black bordered square. The black bordered square is basically to represent the values that the algorithm had gone through, and are sorted. Once the value in the red square has found its place within the values within the black border, the red box for that value changes to black. Then, the next unboxed value will be pulled out the following after.

```
for i      1 to length(A) - 1
    j      i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j      j - 1
    end while
end for
```

codes/insertionSort.java

describe the psuedo code

2.7.9 The Evaluation Design

There are three criteria that will be assessed for the *Algorithm Animation* program. These three refers to the usability of the program, ensuring the the program is easy and learned to use.

Another criteria which applies on the animations of the algorithms of the program needs to be comprehensive for the users.

Of course, when it comes to developing programs that are used to teach, another criteria that comes hand in hand with comprehensiveness would be the correctness of the animation. The animation and the information that comes with them need to most importantly depict the algorithm correctly.

I left it off here

Chapter 3

Review against plan

Todo list