Web-based Demonstration of Pseudo-Random Number Generator RC4

Submitted by: Yao Yawen

Supervisor: Anwitaman Datta

Co-supervisor: Sourav Sen Gupta

School of Computer Science & Engineering

2018

Contents

Abstract

Rivest Cipher 4 is a stream cipher designed in 1987 for RSA Security. Its algorithm uses a key of adjustable length (range of length is from 1 to 256 bytes), and produces pseudo-random bytes through a random permutation. The period of this cipher has been estimated to be above 10^{1000}. For a widely-implemented cipher like RC4, an interactive demonstration that explains its algorithm is however not found available on the Internet. This causes a need for a web demonstration implementation of RC4. In this report, an attempt to build such a web demo is documented. The interactivity in the web demo design demands for sophisticated animations which are challenging to implement. The focus of this report is on the implementations of two particularly complex animations and the animation techniques developed. Flaws and badly-made design choices of the web demo are also discussed in the report.

Acknowledgements

I would like to express my deepest gratitude to my final year project supervisor, Professor Anwitaman Datta, whose guidance and encouragement have supported me through the most difficult time of my project. Without his kindness and supervision this project could not have been completed.

Secondly, I would like to thank my mentor Professor Arvind Easwaran for his help and advices. Lastly, I am particularly grateful for the support provided by my friends and family.

Yao Yawen
November 2018

Terminology

Cryptography	Cryptography is the science or study of techniques of secret writing [1]. It protects messages and communications by encoding them so only authorized users can read and interpret their true meaning. The pre-fix "crypt" means "hidden" or "vault" and the suffix "graphy" stands for "writing" [2]. In computer science, cryptography refers to the use of deterministic encryption algorithms which are derived from mathematical concepts to establish secure data transactions [3].
Encryption	Encryption is the process of transforming human-readable information into unintelligible text. This process uses one or more encryption algorithms and a specified key to encode messages. Decryption is the reverse process.
Plaintext	Plaintext refers to human-readable messages.
Ciphertext	Ciphertext refers to encoded, unintelligible messages. Decryption of ciphertext generates plaintext.
Cipher	A system of writing that most people cannot understand, so that the message is secret; a code [1]. In the digital era, cipher is more widely-known as the algorithm of computerized encryption and decryption. A cipher defines a series of steps or instructions to be followed to encode or decode a message. In many cases, it also uses an encryption or decryption key [3]. The key may be generated by itself, another cipher or environmental variables.
Symmetric Key Encryption	Symmetric key encryption uses the same key for encryption and decryption [4].

Asymmetric Key Encryption	Asymmetric Key Encryption uses a public key, which is readily accessible to anyone, and a private key which is known only to the user. Each public key has a corresponding private key. Data encrypted with a public key can only be decrypted by its corresponding private key, and vice versa. Thus, only the user can read the encrypted message. The user can also leave a digital signature on his data by encoding it with the private key [5].
Steganography	Steganography is a cryptography technique which hides the secret message within other mediums, such as another message, picture or video. The secret message is not transformed, but rather blended into its wrapping medium [6].
Stream Cipher	A stream cipher encrypts a stream of data. Encryption occurs at bit level or byte level. Encryption key varies with every bit or byte.
Block Cipher	A block cipher encrypts a block of data at one time. It outputs an equal-length block of ciphertext. The size of a block may vary in different ciphers. Typically, a block size of 64 or 128 bits is used [7].
Period of a cipher	Period of a cipher is the number of encryption operations a cipher must perform before the output of the cipher repeats.

Acronyms

RC4	Rivest Cipher 4
AES	Advanced Encryption Standard
NTU	Nanyang Technological University
WPA	Wi-Fi Protected Access
HTML5	HyperText Markup Language
CSS	Cascading Style Sheets
DOM	Document Object Model

List of Figures

Chapter 1

Introduction

Cryptography – the science or study of secret writing – has been a revolving topic of interest for thousands of years. The use of encryption techniques dated as far back as 4000 years ago, when an ancient Egyptian scribe used a simple substitution cipher in his drawings on the walls of a tomb. Caesar's cipher, one of the most well-known and easiest encrypting method, was invented by Julius Caesar to protect military intelligence during the time of the Roman Republic **[1]**. Another famous wartime cipher was the Enigma machine. Unlike its 1900-years-older predecessor, the Enigma machine was a sophisticated version of polyalphabetic substitution cipher with 10^{114} possible configurations, therefore extremely difficult to break in its time. Coming into the modern era, the invention of Internet and the widespread of computers have enabled people to perform more and more activities online. As an increasing amount of information has been produced in or sent over the Internet, cryptography has been used to establish secure data transactions.

Modern cryptography mainly consists of symmetric key encryption, asymmetric key encryption and steganography **[1]**. In symmetric key encryption, the message sender and the recipient share one key, also called private key, for enciphering and deciphering. The private key must be exchanged prior any data transaction **[8]**. Asymmetric key encryption, also called public-key cryptography, was invented in the 1970's because people needed a method to exchange the secret key safely without meeting each other in the physical world **[5]**. Steganography is the art of concealing a message within another message in plain sight.

Over the years, many symmetric key encryption algorithms have been developed to protect privacy and commerce. They mostly fall into two categories: stream cipher and block cipher. The main difference between them is that encryption occurs at bit level or byte level in a stream cipher, while in a block cipher it occurs at block level -- the size of a block of data depends on the exact block cipher used. A stream cipher takes in a stream of plaintext and outputs a stream of ciphertext dynamically. Examples of stream ciphers are one-time-pad, Linear Feedback Shift Register and RC4. A block cipher encrypts one chunk of data at once, and then outputs an equal-length block. Typically, a block size of 64 or 128 bits is used. Blowfish, RC5 and AES are some popular block ciphers **[9]**.

In computer science, cryptography has been an important topic of study and research. Many academical institutions have opened courses about cryptography alone, or about cybersecurity in which cryptography is inevitably introduced to the students. In NTU, the School of Computer Science and Engineering offers course CZ/CE4024 – Cryptography and Network Security to undergraduates who are interested in how basic cryptography algorithms work and their application in real-world information security systems **[10]**. Some ciphers mentioned before, such as AES and RC4, are also explained in detail in this course. To facilitate the understanding of the basic enciphering algorithms and illustrate how specific ciphers work, this course has web demonstrations of various ciphers **[11]**. These web demos transform text descriptions of ciphers into interactive animations, and give users the opportunity to participate in each step of encryption. However, not every encryption algorithm taught in this course has a web demo. One example is RC4, which is simple yet efficient – a good example of stream cipher design. This caused need for its web demo implementation.

This report documents the implementation process of the RC4 web demo for course CZ/CE4024 – Cryptography and Network Security. Target users of this web demo are computer science and engineering undergraduates. It is assumed that users understand basic computer science and mathematical concepts such as vector, pseudocode and encryption key. The report explores the mathematics used in RC4 design, justifies the true randomness of encryption key generated by RC4, and explains the functionality of the web demo and how it has been written using HTML, JavaScript, jQuery and Web Animations API. Limitations and potentials of web animation techniques touched in the demo are analyzed as well.

1.1 Organisations

The body of this report includes five chapters. Chapter 1 introduces the background information of RC4 and explains the motivations behind its web demo implementation. Chapter 2 describes the inner workings of RC4 in detail and reviews past works done to illustrate RC4 in graphics and animations. Chapter 3 walks the reader through the animation implementations step by step and shares insights on its limitations and room for improvement. Chapter 3 also introduces a range of animation techniques provided in CSS, jQuery, JavaScript and Web Animation API. Chapter 4 concludes the report and shares recommendations.

Chapter 2

Literature Review

2.1 RC4 algorithm

RC4 is a stream cipher designed for RSA Security and named after its designer Ron Rivest. Its algorithm uses a key of adjustable length (range of length is from 1 to 256 bytes), and produces pseudo-random bytes through a random permutation. The period of this cipher has been estimated to be above 10^{100}. RC4 is remarkably simple and runs very quickly in software. Due to its simplicity and high speed, it has become widely-used in many applications, such as Wi-Fi Protected Access (WPA) and Kerberos **[9]**.

There are two parts of RC4: Initiation of S, and Stream Generation. In the first part, state vector S and a temporary vector T are initialized as shown in the pseudocode:

```
for i = 0 to 255 do
    S[i] = i;
    T[i] = K[i mod keylen];
```

K stands for key. It is a user-input key and has an adjustable length. Elements of K are used to fill in the elements of vector T. If K is shorter than 256 bytes, it would be repeated as many times as needed to fill out T. Values of vector S is initialized in an ascending order in the range of 0 to 255: S[0] = 0, S[1] = 1, …, S[254] = 254, S[255] = 255.

Next, vector S is permuted according to values of vector T. K is no longer useful in subsequent operations. The permutation pseudocode is presented below:

```
j = 0;
for i = 0 to 255 do
    j = (j + S[i] + T[i]) mod 256;
    Swap (S[i], S[j]);
```

Starting from S[0] to S[255], each vector S element is swapped with another S element the index of which is the value of j. The value of j is determined by vector T.

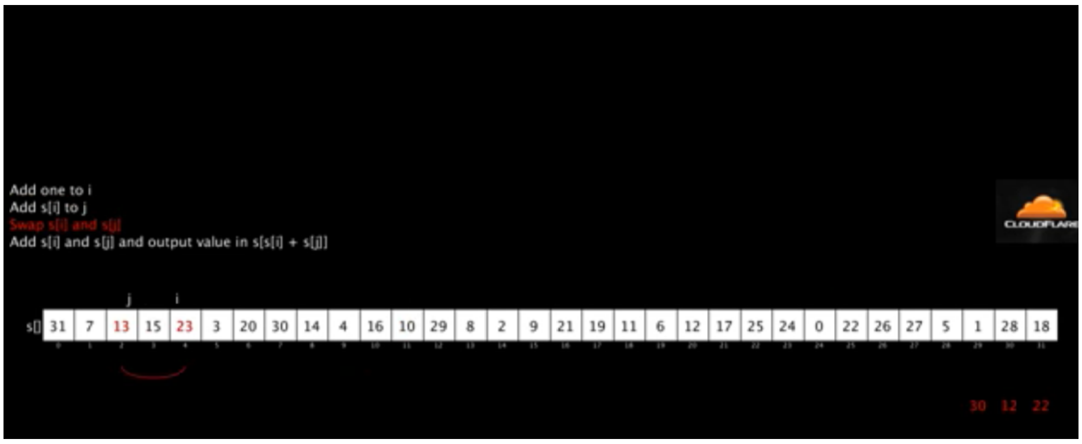
The second part generates a byte k by selecting one element from the 255 elements of S. Afterwards S is permutated once so that the next round of k-selection uses a different S. The pseudocode below illustrates the permutation:

```
i, j = 0;
while (true)
    i = (i + 1) mod 256;
    j = (j + S[i]) mod 256;
    swap (S[i], S[j]);
    t = (S[i] + S[j]) mod 256;
    k = S[t];
```

It starts with the permuted vector S from the second stage. The cipher loops infinitely through vector S. For each S[i], a j value is calculated according to its value and the value of the previous j. S[i] and S[j] are then swapped, changing the configuration of vector S. Sum of values of S[i] and S[j] is used as an index which points another element in vector S. Value of this element S[S[i] + S[j]], is the pseudo-random byte output of RC4.

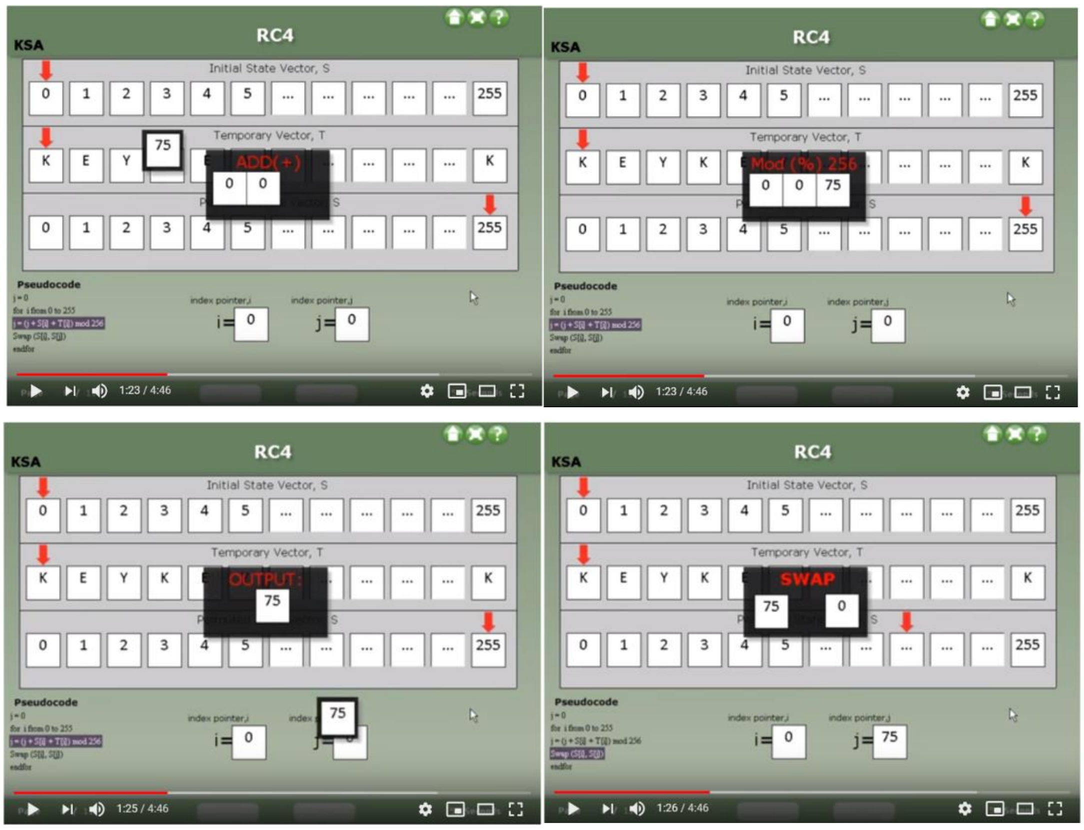
2.2 Past works done to visualize RC4

There are a few visualizations of RC4 found on the Internet. They are either in the form of videos or in the form of applications. Below are screenshots of two RC4 simulation videos:



Add one to i
Add s[i] to j
Swap s[i] and s[j]
Add s[i] and s[j] and output value in s[s[i] + s[j]]

Figure 1. YouTube video "RC4 Algorithm" showing the 'swap' operation of RC4 algorithm [12]

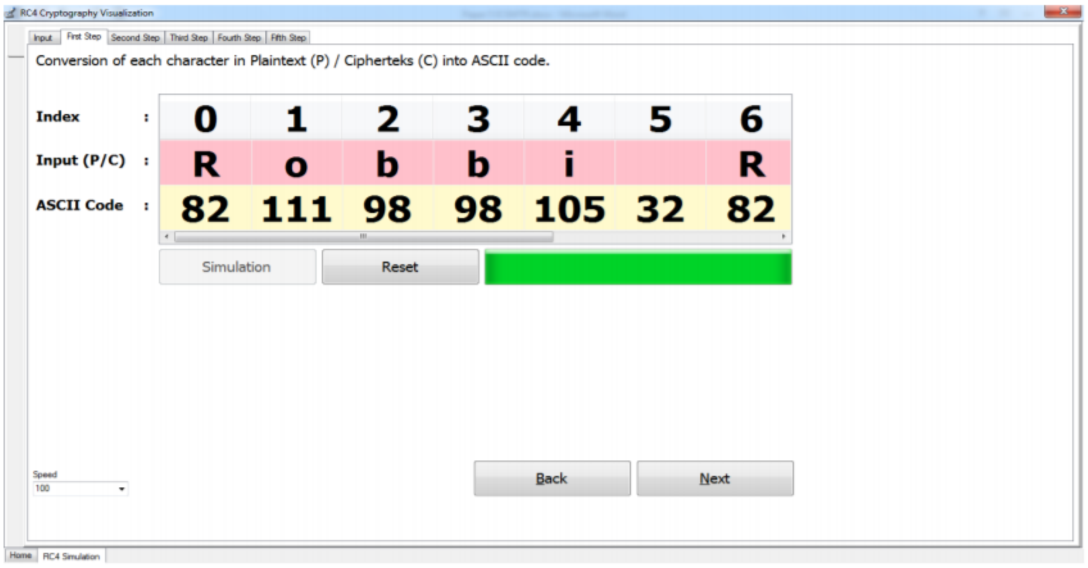


```
Pseudocode
j=0
for i from 0 to 255
j=(j+S[i]+T[i]) mod 256
Swap (S[i], S[j])
endfor
```

Figure 2. YouTube video named "RC4 security algorithm" [13]

In general, these videos display the pseudocode of RC4 operation on one side and put animation section in the center. For each step, the corresponding pseudocode is highlighted in a different color, and then the operation is performed. Figure 2 is an example of a video showing the four steps of the Initial Permutation of vector S. Video format of RC4 visualization is straightforward and easy to understand. However, its encryption key value and plaintext is prefixed, lacking interactivity.

The other format is application. Two applications were found to truly simulate every step of RC4 algorithm. One application was developed by Vishwas Gagrani (YouTube account name) [13]. the video shown in Figure 2 is an introductory recording of his application with a given key and plaintext. This application was built using FLASH. It used to be freely available on Google Play but was removed at some point of time in 2018 [13]. Another application was never made available to the mass, only presented in a conference paper [14]. It was written in Visual Basic.Net 2008 programming language. In addition to RC4 algorithm, this visualization model also provides ASCII conversion of key and plaintext. Figure 3 shows this function:



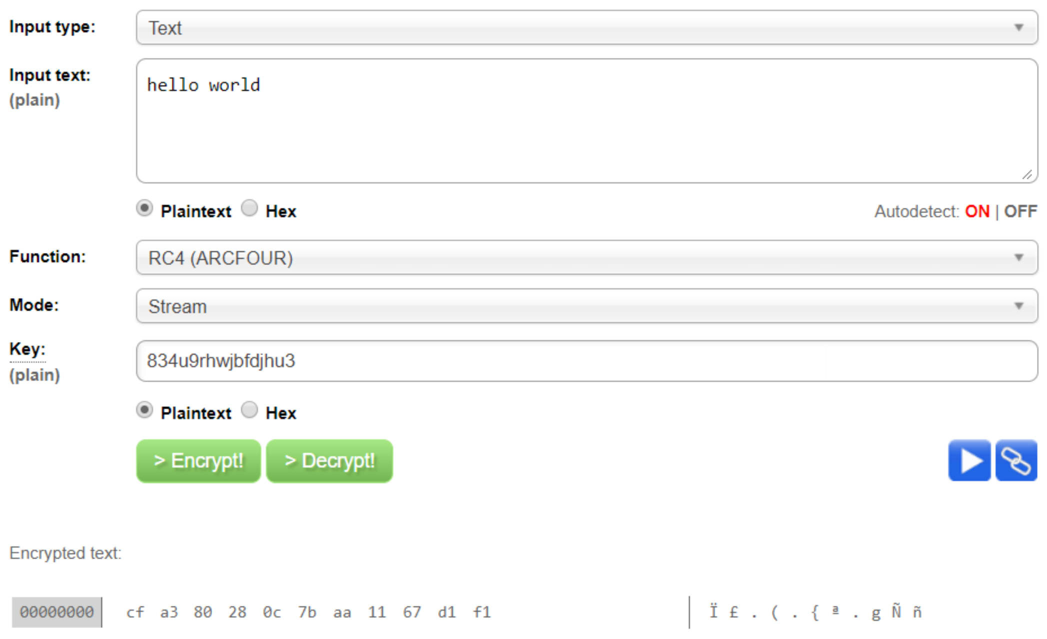
Conversion of each character in Plaintext (P) / Cipherteks (C) into ASCII code.

Index :	0	1	2	3	4	5	6
Input (P/C) :	R	o	b	b	i		R
ASCII Code :	82	111	98	98	105	32	82

Figure 3. ASCII conversion of plaintext

Other than videos and applications, there are many webpage simulations of RC4. However, they do not show the internal operations between input and output.

Figure 4 below gives an example.



Input type: Text

Input text: (plain)

```
hello world
```

[✓ Plaintext] Hex   Autodetect: ON | OFF

Function: RC4 (ARCFOUR)

Mode: Stream

Key: (plain) 834u9rhwjbfdjhu3

[✓ Plaintext] Hex

Encrypted text:

```
00000000  cf a3 80 28 0c 7b aa 11 67 d1 f1        | Ï £ . ( . { ª . g Ñ ñ
```

Figure 4. Webpage simulation of RC4 [16]

To summarize past works done on RC4 visualization, some videos and applications are helpful RC4 simulations for people who want to gain a solid, detailed understanding of RC4. Unfortunately, the videos are non-interactive with little explanation to aid learning, and the applications are not accessible to the NTU community. This caused a need for an interactive RC4 simulation to be developed and made available to not only NTU students but also everyone who are interested in cryptography.

Chapter 3

Implementation of RC4 Web Demonstration

3.1 General structure

The web demo (demonstration) has four sections: Introduction, Stage 1, Stage 2 and Stage3.

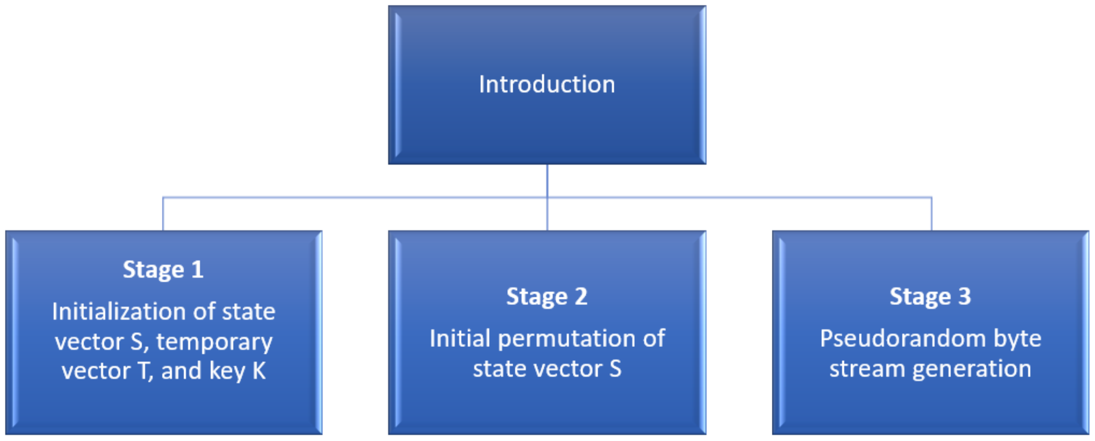
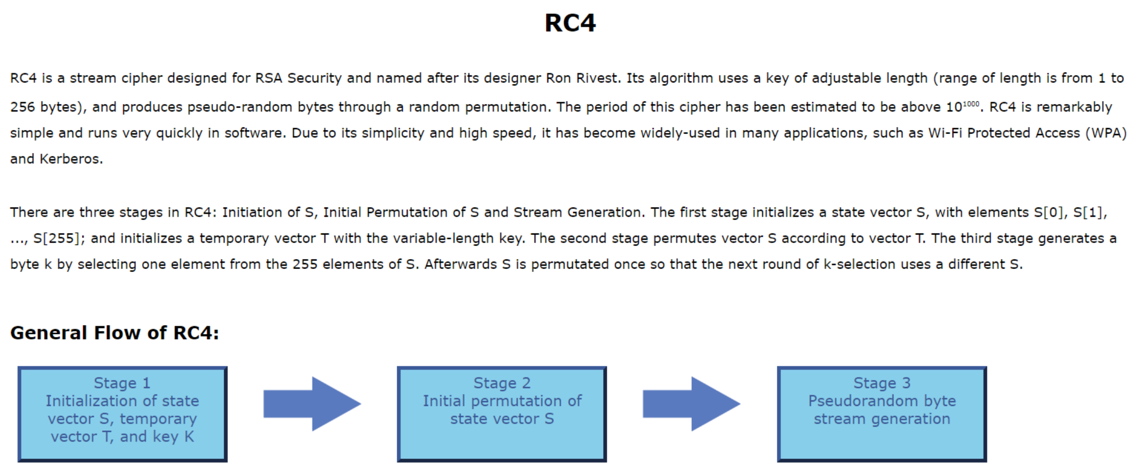

Introduction
Stage 1
Initialization of state vector S, temporary vector T, and key K
Stage 2
Initial permutation of state vector S
Stage 3
Pseudorandom byte stream generation

Figure 5. Structure of web demo

Introduction section gives the reader a brief overview of RC4. RC4 algorithms can be divided into three sub-sections, hence the three stages. The division of RC4 process is not an industrial practice, but rather a choice made during implementation in order to make RC4 easier to comprehend. The first two stages correspond to the first part described in 2.1 RC4 algorithm; the third stage corresponds to the second part. Below are screenshots of the four sections.



RC4

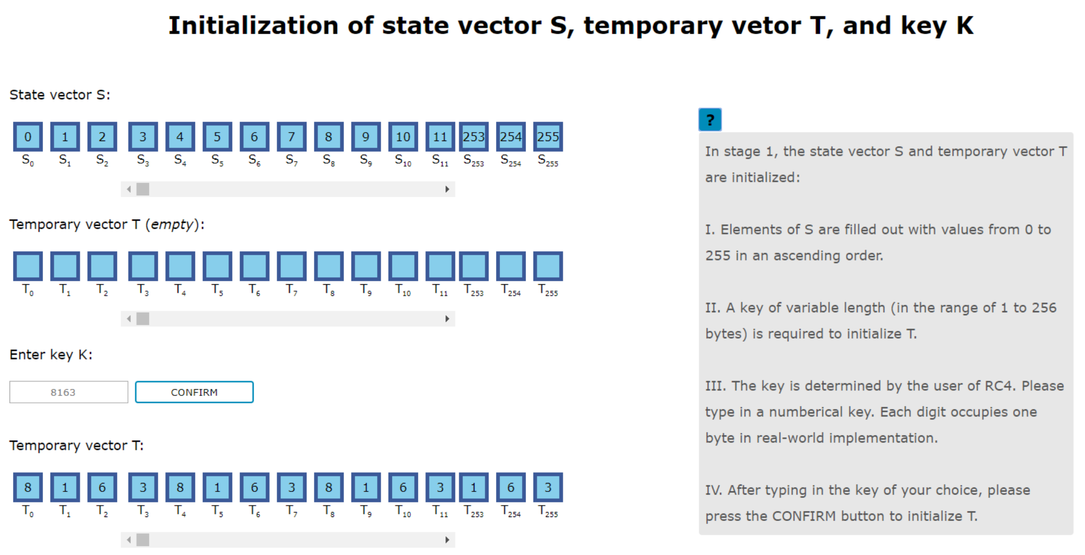
RC4 is a stream cipher designed for RSA Security and named after its designer Ron Rivest. Its algorithm uses a key of adjustable length (range of length is from 1 to 256 bytes), and produces pseudo-random bytes through a random permutation. The period of this cipher has been estimated to be above 10^{1000}. RC4 is remarkably simple and runs very quickly in software. Due to its simplicity and high speed, it has become widely-used in many applications, such as Wi-Fi Protected Access (WPA) and Kerberos.

There are three stages in RC4: Initiation of S, Initial Permutation of S and Stream Generation. The first stage initializes a state vector S, with elements S[0], S[1], ..., S[255]; and initializes a temporary vector T with the variable-length key. The second stage permutes vector S according to vector T. The third stage generates a byte k by selecting one element from the 255 elements of S. Afterwards S is permutated once so that the next round of k-selection uses a different S.

General Flow of RC4:

Stage 1
Initialization of state vector S, temporary vector T, and key K
Stage 2
Initial permutation of state vector S
Stage 3
Pseudorandom byte stream generation

Figure 6. Web demo section 1



Initialization of state vector S, temporary vetor T, and key K

State vector S:

0	1	2	3	4	5	6	7	8	9	10	11	253	254	255
S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{253}	S_{254}	S_{255}

Temporary vector T (*empty*):

T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{253}	T_{254}	T_{255}

Enter key K:

8163

Temporary vector T:

8	1	6	3	8	1	6	3	8	1	6	3	1	6	3
T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{253}	T_{254}	T_{255}

In stage 1, the state vector S and temporary vector T are initialized:

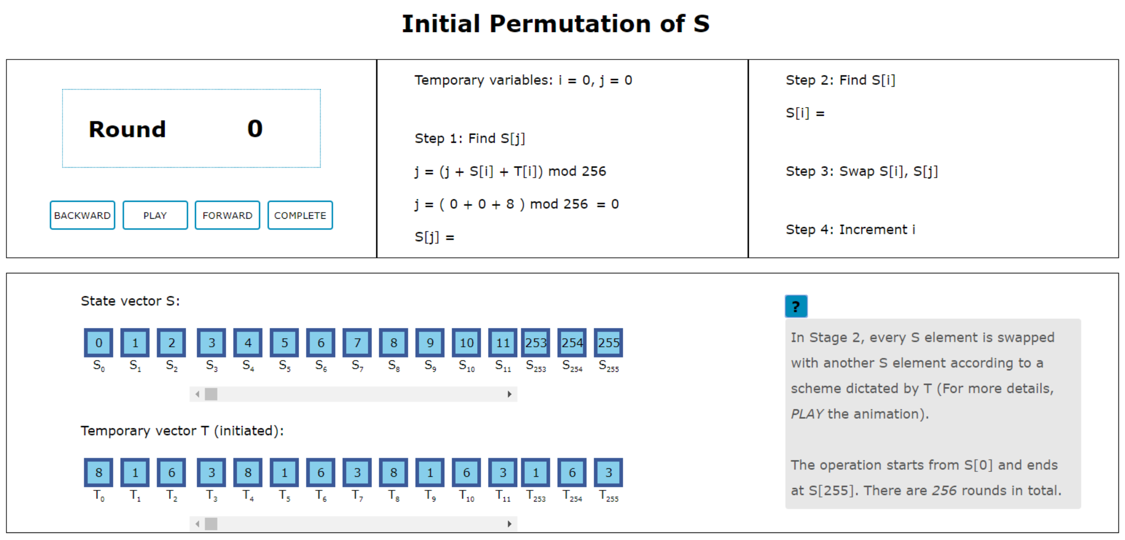
I. Elements of S are filled out with values from 0 to 255 in an ascending order.

II. A key of variable length (in the range of 1 to 256 bytes) is required to initialize T.

III. The key is determined by the user of RC4. Please type in a numberical key. Each digit occupies one byte in real-world implementation.

IV. After typing in the key of your choice, please press the CONFIRM button to initialize T.

Figure 7. Web demo section 2



Initial Permutation of S

Round 0

Temporary variables: i = 0, j = 0

Step 1: Find S[j]

j = (j + S[i] + T[i]) mod 256

j = (0 + 0 + 8) mod 256 = 0

S[j] =

Step 2: Find S[i]

S[i] =

Step 3: Swap S[i], S[j]

Step 4: Increment i

State vector S:

0	1	2	3	4	5	6	7	8	9	10	11	253	254	255
S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{253}	S_{254}	S_{255}

Temporary vector T (initiated):

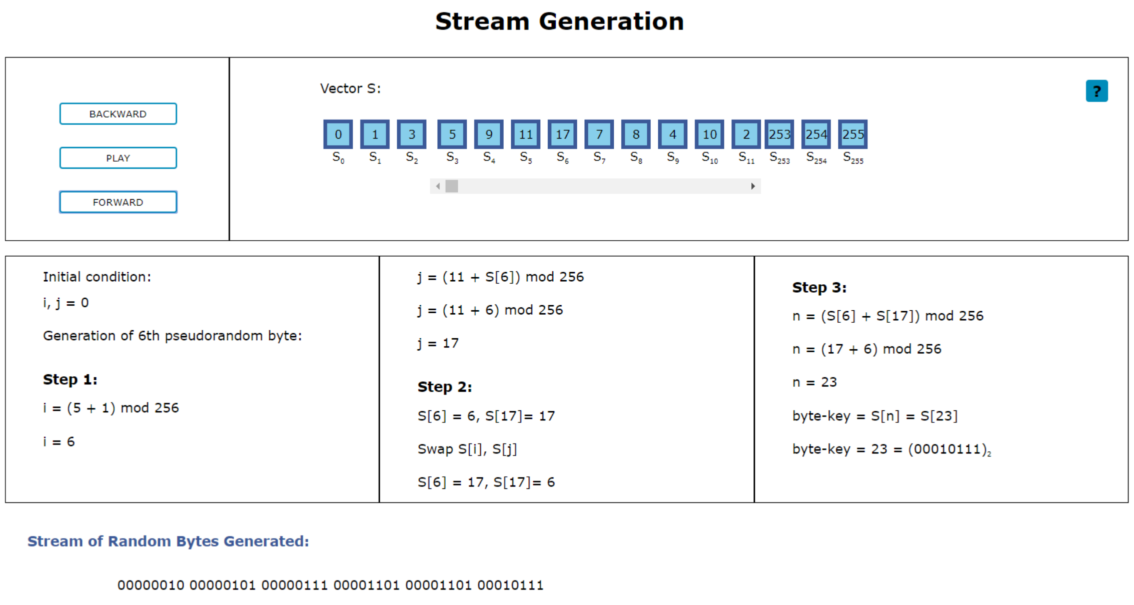
8	1	6	3	8	1	6	3	8	1	6	3	1	6	3
T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{253}	T_{254}	T_{255}

?

In Stage 2, every S element is swapped with another S element according to a scheme dictated by T (For more details, *PLAY* the animation).

The operation starts from S[0] and ends at S[255]. There are *256* rounds in total.

Figure 8. Web demo section 3



Stream Generation

Vector S:

0	1	3	5	9	11	17	7	8	4	10	2	253	254	255
S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{253}	S_{254}	S_{255}

Initial condition:

i, j = 0

Generation of 6th pseudorandom byte:

Step 1:

i = (5 + 1) mod 256

i = 6

j = (11 + S[6]) mod 256

j = (11 + 6) mod 256

j = 17

Step 2:

S[6] = 6, S[17]= 17

Swap S[i], S[j]

S[6] = 17, S[17]= 6

Step 3:

n = (S[6] + S[17]) mod 256

n = (17 + 6) mod 256

n = 23

byte-key = S[n] = S[23]

byte-key = 23 = $(00010111)_2$

Stream of Random Bytes Generated:

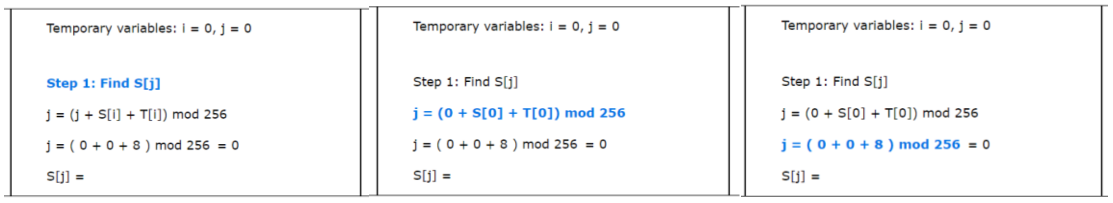
00000010 00000101 00000111 00001101 00001101 00010111

Figure 9. Web demo section 4

The text and its styling in the webpage are created using HTML5 and CSS. An external CSS style sheet is used to keep the webpage document clean and simple, while making it easy to control its overall CSS style. The animations are implemented mainly in JavaScript and jQuery. Using basic animation methods offered by CSS, JavaScript and jQuery as building blocks, complex animations can be realized.

3.2 Sequential Web Animation Implementation

In stage 2 and 3, the algorithm steps written in pseudo-code are highlighted and made bold in sequence. Figure 10 illustrates the sequential animation:



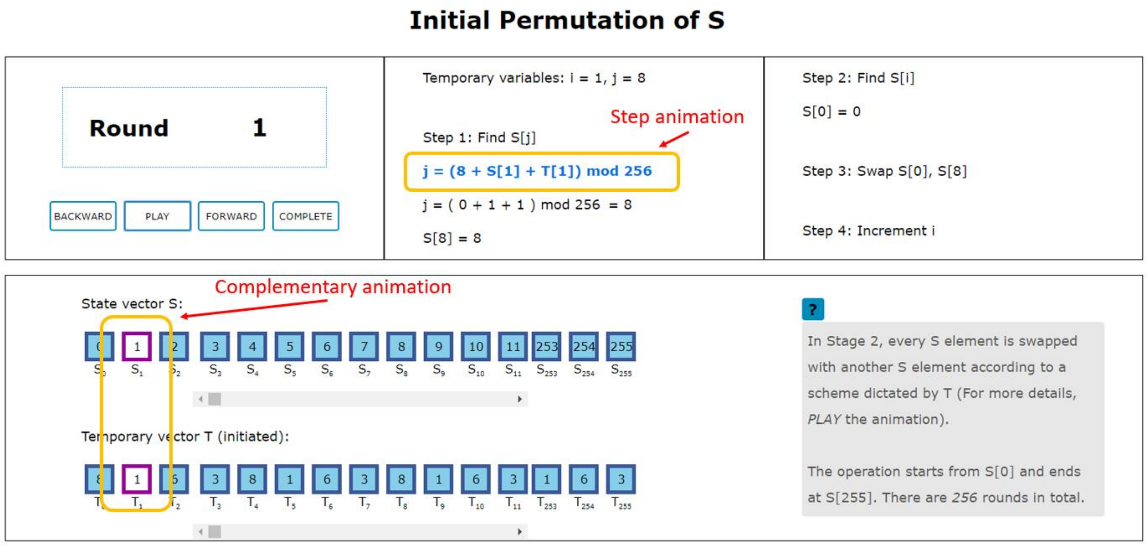
```
Temporary variables: i = 0, j = 0

Step 1: Find S[j]
j = (j + S[i] + T[i]) mod 256
j = ( 0 + 0 + 8 ) mod 256  = 0
S[j] =
```

```
Temporary variables: i = 0, j = 0

Step 1: Find S[j]
j = (0 + S[0] + T[0]) mod 256
j = ( 0 + 0 + 8 ) mod 256  = 0
S[j] =
```

```
Temporary variables: i = 0, j = 0

Step 1: Find S[j]
j = (0 + S[0] + T[0]) mod 256
j = ( 0 + 0 + 8 ) mod 256  = 0
S[j] =
```

Figure 10. Stage 2 sequential animations

While the steps are being animated, other elements in the same stage may have complementary animations, as shown in Figure 11:



```
Temporary variables: i = 1, j = 8

Step 1: Find S[j]
j = (8 + S[1] + T[1]) mod 256
j = ( 0 + 1 + 1 ) mod 256  = 8
S[8] = 8
```

```
Step 2: Find S[i]
S[0] = 0

Step 3: Swap S[0], S[8]

Step 4: Increment i
```

In Stage 2, every S element is swapped with another S element according to a scheme dictated by T (For more details, *PLAY* the animation).

The operation starts from S[0] and ends at S[255]. There are *256* rounds in total.

Figure 11. Stage 2 concurrent animations

Moreover, the animations can be paused and reactivated using the ‘PLAY’ button shown in the upper left rectangular section of Figure 11. Pressing ‘PLAY’ button triggers the animation. The text within the button changes to ‘PAUSE’. Pressing this button again pauses the animation and the text reverts to ‘PLAY’.

The Stage 2 sequence of animations ends at step 4. The temporary variable i and the round counter are incremented, and the sequence of animations restarts again. The Stage 2 section would cycle through this animation sequence over and over until variable i is incremented to its upper limit – 256.

This chapter explains how the sequential animations mentioned above were implemented using CSS animations and JavaScript. The implementation of sequential animations consists of three parts:

1. Implementation of toggling animations on and off
2. Implementation of iterating a series of animations infinitely
3. Implementation of animations starting and finishing in a smooth sequence

3.2.1 CSS Animation and JavaScript

CSS animations reply on predefined keyframes to animate HTML elements. One example would be the animation of the round counter number in Stage 2. Figure 12 illustrates the animation:

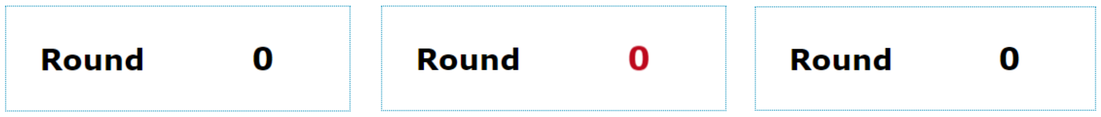
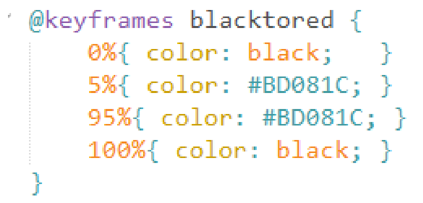


Figure 12. Animation of the round counter in stage 2

To realize this animation, an @keyframes animation that defines this behavior is created in the CSS style sheet.

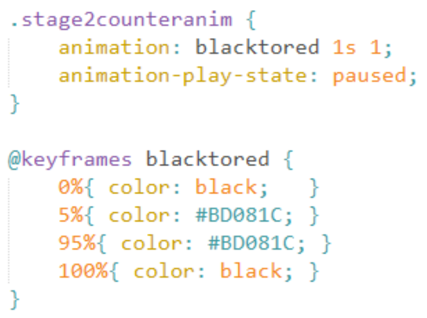


```
@keyframes blacktored {
    0%{ color: black;   }
    5%{ color: #BD081C; }
    95%{ color: #BD081C; }
    100%{ color: black; }
}
```

Figure 13. @Keyframes "blacktored"

This defines a 4-stage animation named "blacktored". At 0%, CSS properties defined in the brackets are to be the initial CSS properties of the element before the animation starts. From 0% to when the animation is 5% complete, the CSS properties change gradually, making the counter number turns from black to red. From 5% to 95% of the time duration, the CSS properties remain the same, therefore the counter number stays in red color. At 100% which is the end state of the animation, the counter number turns black.

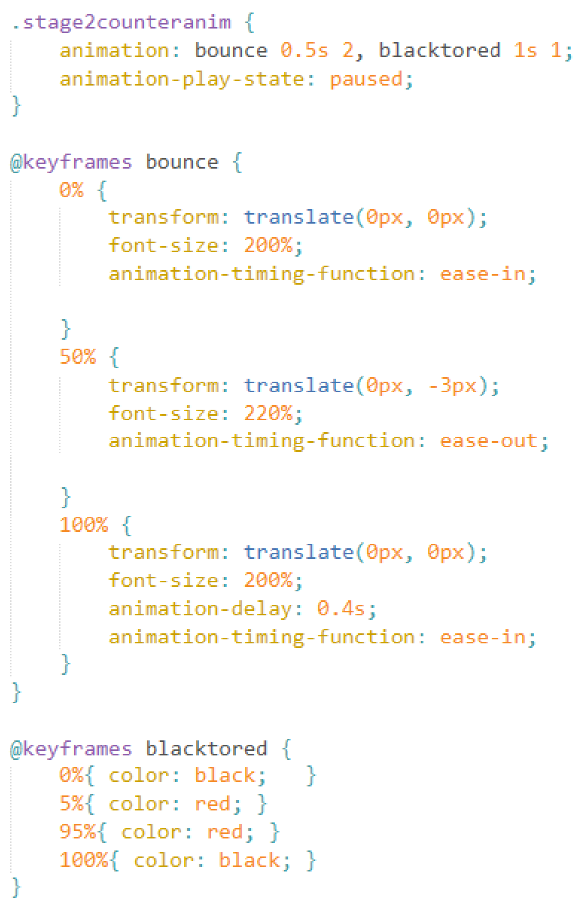
Animation "blacktored" is bound to desired elements using the CSS property "animation". "1s" specifies the duration. "1" specifies the number of times the animation is iterated.



```
.stage2counteranim {
    animation: blacktored 1s 1;
    animation-play-state: paused;
}

@keyframes blacktored {
    0%{ color: black;   }
    5%{ color: #BD081C; }
    95%{ color: #BD081C; }
    100%{ color: black; }
}
```

Figure 14. Bind an animation to elements of class "stage2counteranim"

In the web demo, the counter number not only changes its color, but also bounces twice because the CSS property “animation” can accept more than one animation, as shown in the code section below:

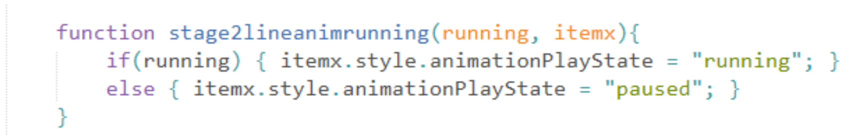


```
.stage2counteranim {
    animation: bounce 0.5s 2, blacktored 1s 1;
    animation-play-state: paused;
}

@keyframes bounce {
    0% {
        transform: translate(0px, 0px);
        font-size: 200%;
        animation-timing-function: ease-in;

    }
    50% {
        transform: translate(0px, -3px);
        font-size: 220%;
        animation-timing-function: ease-out;

    }
    100% {
        transform: translate(0px, 0px);
        font-size: 200%;
        animation-delay: 0.4s;
        animation-timing-function: ease-in;
    }
}

@keyframes blacktored {
    0%{ color: black;   }
    5%{ color: red; }
    95%{ color: red; }
    100%{ color: black; }
}
```

Figure 15. Code section of stage 2 counter animations

3.2.2 Implementation of Toggling Animations on and off

In Figure 15, the "animation-play-state" is set to "paused", because its default value is "running" which would cause the animation to run immediately after the web page is fully loaded. This is not desirable as the web demo needs the animations to run only when the user triggers it. By setting the "animation-play-state" to "running" when the user presses "PLAY" button and setting the "animation-play-state" to "paused" when the user presses "PAUSE" button, CSS animations can be paused and activated anytime. This is realized through the JavaScript function shown below:

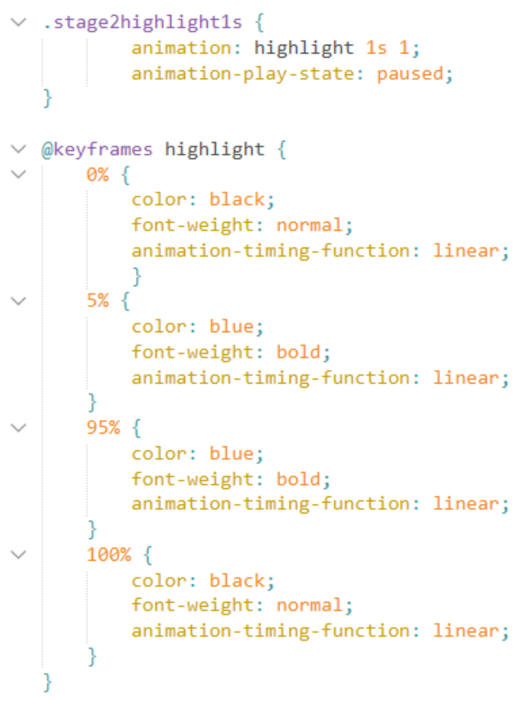


```
function stage2lineanimrunning(running, itemx){
    if(running) { itemx.style.animationPlayState = "running"; }
    else { itemx.style.animationPlayState = "paused"; }
}
```

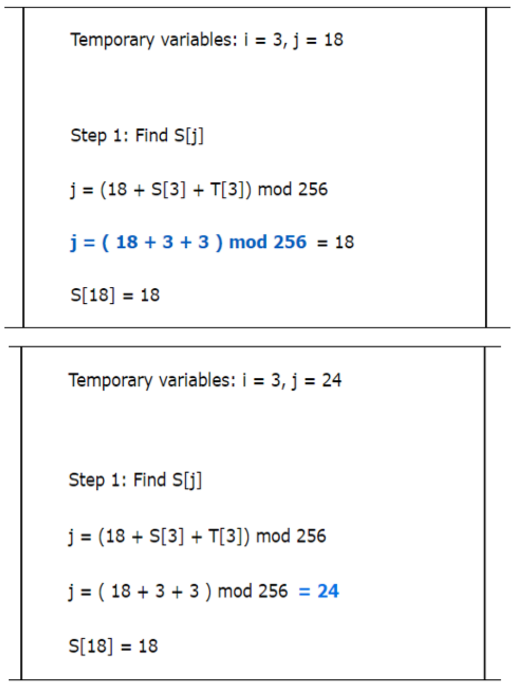
Figure 16. Function that changes "animation-play-state" property

In Figure 16, function stage2lineanimrunning(running, itemx) takes in a Boolean parameter "running", and a HTML element "itemx". If "running" is equal to True, "animation-play-state" of element "itemx" is set to "running". If not, "animation-play-state" is set to "paused".

3.2.3 Implementation of Infinite Iterations Through A Series of Animations



```
.stage2highlight1s {
        animation: highlight 1s 1;
        animation-play-state: paused;
}

@keyframes highlight {
    0% {
        color: black;
        font-weight: normal;
        animation-timing-function: linear;
        }
    5% {
        color: blue;
        font-weight: bold;
        animation-timing-function: linear;
    }
    95% {
        color: blue;
        font-weight: bold;
        animation-timing-function: linear;
    }
    100% {
        color: black;
        font-weight: normal;
        animation-timing-function: linear;
    }
}
```



Temporary variables: i = 3, j = 18

Step 1: Find S[j]

j = (18 + S[3] + T[3]) mod 256

j = (18 + 3 + 3) mod 256 = 18

S[18] = 18

Temporary variables: i = 3, j = 24

Step 1: Find S[j]

j = (18 + S[3] + T[3]) mod 256

j = (18 + 3 + 3) mod 256 **= 24**

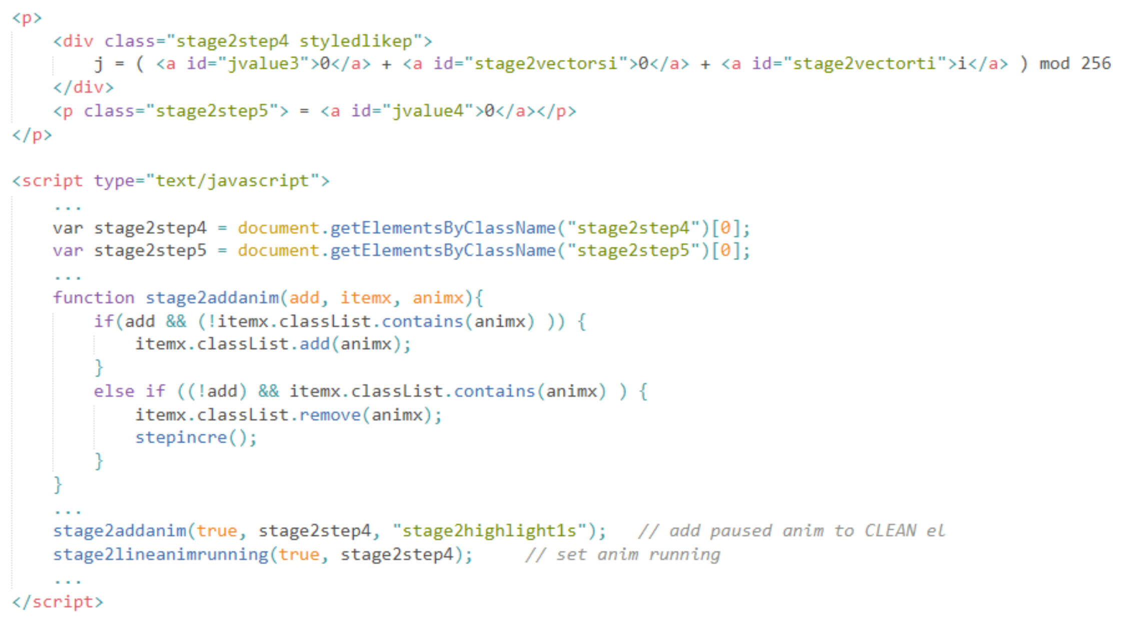
S[18] = 18

Figure 17. CSS animation “highlight”.

Figure 18. Stage 2 steps highlighted in blue sequentially

In the web demo, each pseudo-code step is highlighted in blue for a few seconds in each round. Figure 17 shows the CSS animation code corresponding to this animation as illustrated in Figure 18. Number of iterations of animation “highlight” is 1, therefore this animation highlights its target element once, and ends. It would not run again however its “animation-play-state” changes. The next round would not see any blue-highlighting effect on the target element. However, if the number of iterations of this animation is set to “infinite”, the target element would be switching its color between blue and black continuously. More importantly, animations scheduled after it would never have a chance to run. This issue arises from the design of CSS animation and is not found to be solvable using CSS alone. Therefore, JavaScript is used to tackle this issue.

JavaScript code can circumvent this problem by removing or adding the class that binds the target HTML element with the animation, from the list of classes of that element. Take the class value “stage2highlight1s” in Figure 17 and HTML in Figure 18 as an example.



```
<p>
    <div class="stage2step4 styledlikep">
        j = ( <a id="jvalue3">0</a> + <a id="stage2vectorsi">0</a> + <a id="stage2vectorti">i</a> ) mod 256
    </div>
    <p class="stage2step5"> = <a id="jvalue4">0</a></p>
</p>

<script type="text/javascript">
    ...
    var stage2step4 = document.getElementsByClassName("stage2step4")[0];
    var stage2step5 = document.getElementsByClassName("stage2step5")[0];
    ...
    function stage2addanim(add, itemx, animx){
        if(add && (!itemx.classList.contains(animx) )) {
            itemx.classList.add(animx);
        }
        else if ((!add) && itemx.classList.contains(animx) ) {
            itemx.classList.remove(animx);
            stepincre();
        }
    }
    ...
    stage2addanim(true, stage2step4, "stage2highlight1s");   // add paused anim to CLEAN el
    stage2lineanimrunning(true, stage2step4);     // set anim running
    ...
</script>
```

Figure 19. HTML and JavaScript codes of adding and removing class

In the JavaScript, a third-party library classList is used [16]. It makes addition, removal or toggle of CSS classes easier and more straightforward.

The <div class = "stage2step4 styledlikep"> element is the blue-highlighted line of pseudo-code in the upper box in Figure 18. JavaScript accesses this element with the line “var stage2step4 = document.getElementsByClassName("stage2step4")[0];”. Calling function stage2addanim(true, stage2step4, "stage2highlight1s") changes the list of classes from class = “stage2step4 styledlikep” to class = “stage2step4 styledlikep stage2highlight1s”. The animation would not run because its “animation-play-state” is still “paused”. Function stage2lineanimrunning(true, stage2step4) is called to set the animation in action (the function is shown in Figure 16). After the animation finishes, function stage2addanim(false, stage2step4, "stage2highlight1s") is called to remove the animation class. The animation properties such as number of iterations no longer apply to element stage2step4. In the next round, the same animation class is added to the element, and the number of iterations of animation would be 1 again.

In conclusion, infinite iteration of a sequence of pausable animations can be implemented by adding or removing the animation classes on the target elements.

3.2.4 Implementation of Animations Starting and Ending in A Smooth Sequence

The most common approach to the problem of sequencing animations is to use setTimeout() function:

```
setTimeout(code, millisec);
```

In which the next animation is set running by the code in the parentheses, at the time when “millisec” number of milliseconds has just passed. However, if the user pauses the current animation for a variable length of time, the setTimeout() function would take the pause duration into account. If the pause duration is longer than the timeout period, the next animation would start running while the current animation is not yet complete. This would make the web demo disordered and confusing. Hence the setTimeout() function is not used in the implementation.

Further research found that “the animationend event occurs when a CSS animation has completed” [17] [18]. The animationend event is therefore used with event listener to signal the completion of one animation and trigger the activation of the next one.

In the web demo, the animation sequence control function in stage 2 is pauserunanim(). For stage 3, it is function stage3animation(), which has a similar structure. To avoid redundancy, only pauserunanim() is discussed in the report. Below is its simplified flow:

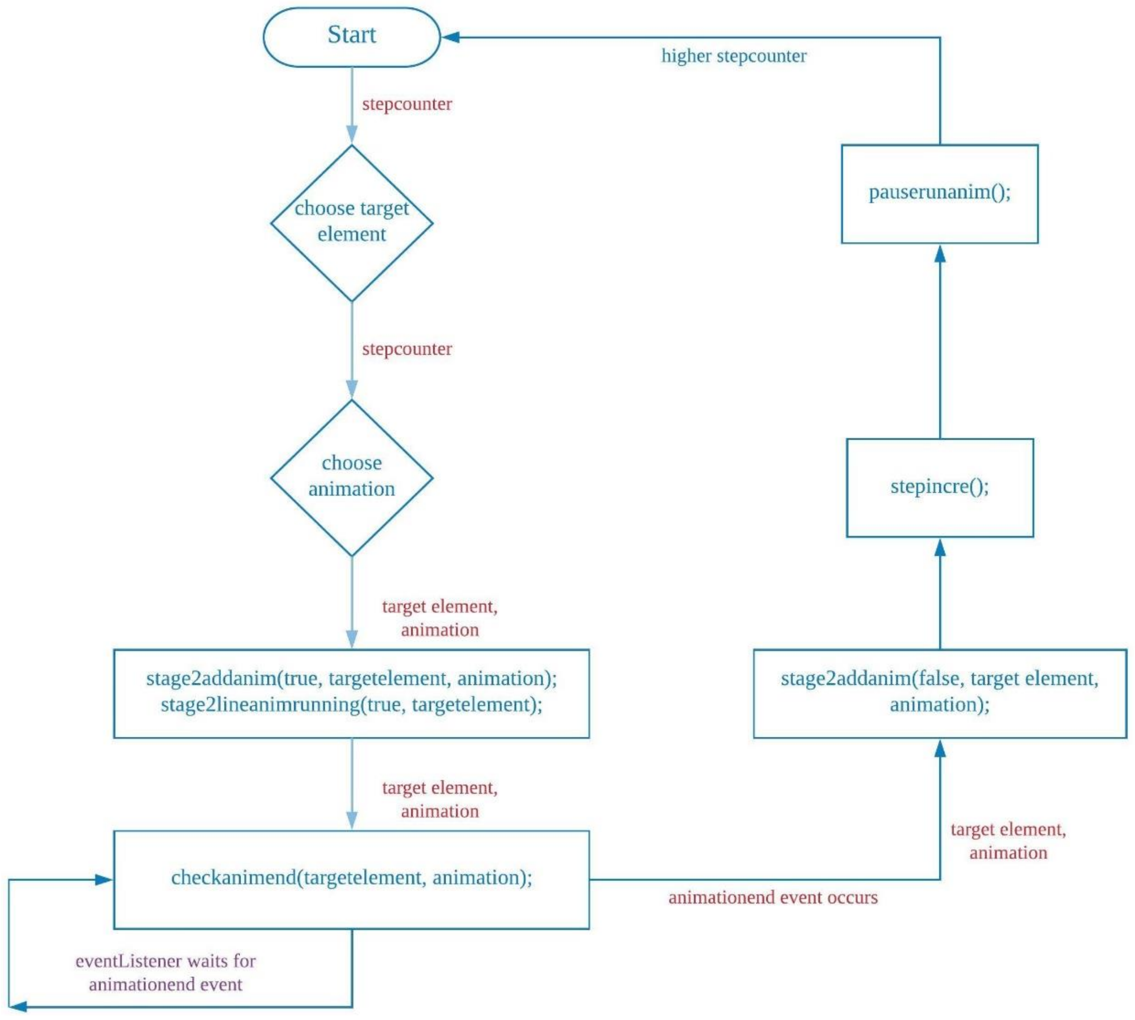
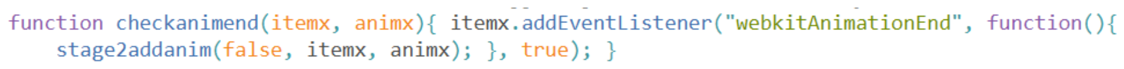

Start
higher stepcounter
stepcounter
choose target element
pauserunanim();
stepcounter
choose animation
stepincre();
target element, animation
stage2addanim(true, targetelement, animation); stage2lineanimrunning(true, targetelement);
stage2addanim(false, target element, animation);
target element, animation
target element, animation
checkanimend(targetelement, animation);
animationend event occurs
eventListener waits for animationend event

Figure 20. Simplified flow of animation sequence control function pauserunanim()

The global variable stepcounter keeps track of which animation step the web demo is at. The flow of pauserunanim() is explained in detail below:

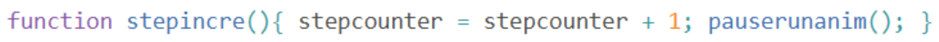
1. The function pauserunanim() uses stepcounter to select the corresponding target element and animation to run.

2. It then calls stage2addanim(true, targetelement, animation) and passes the target element as targetelement parameter, and the animation as animation parameter into the animation-adding-or-removing function.
3. As the Boolean parameter is set to true, stage2addanim() will perform animation class addition.
4. stage2lineanimrunning(true, targetelement) is then called to modify the “animation-play-state” to be “running”. At this point of time, the animation would start running.
5. The control function pauserunanim() then moves on to call checkanimend(targetelement, animation). This function creates an EventListener on “animationend” event on the target element. Below is its JavaScript code:



```
function checkanimend(itemx, animx){ itemx.addEventListener("webkitAnimationEnd", function(){
    stage2addanim(false, itemx, animx); }, true); }
```

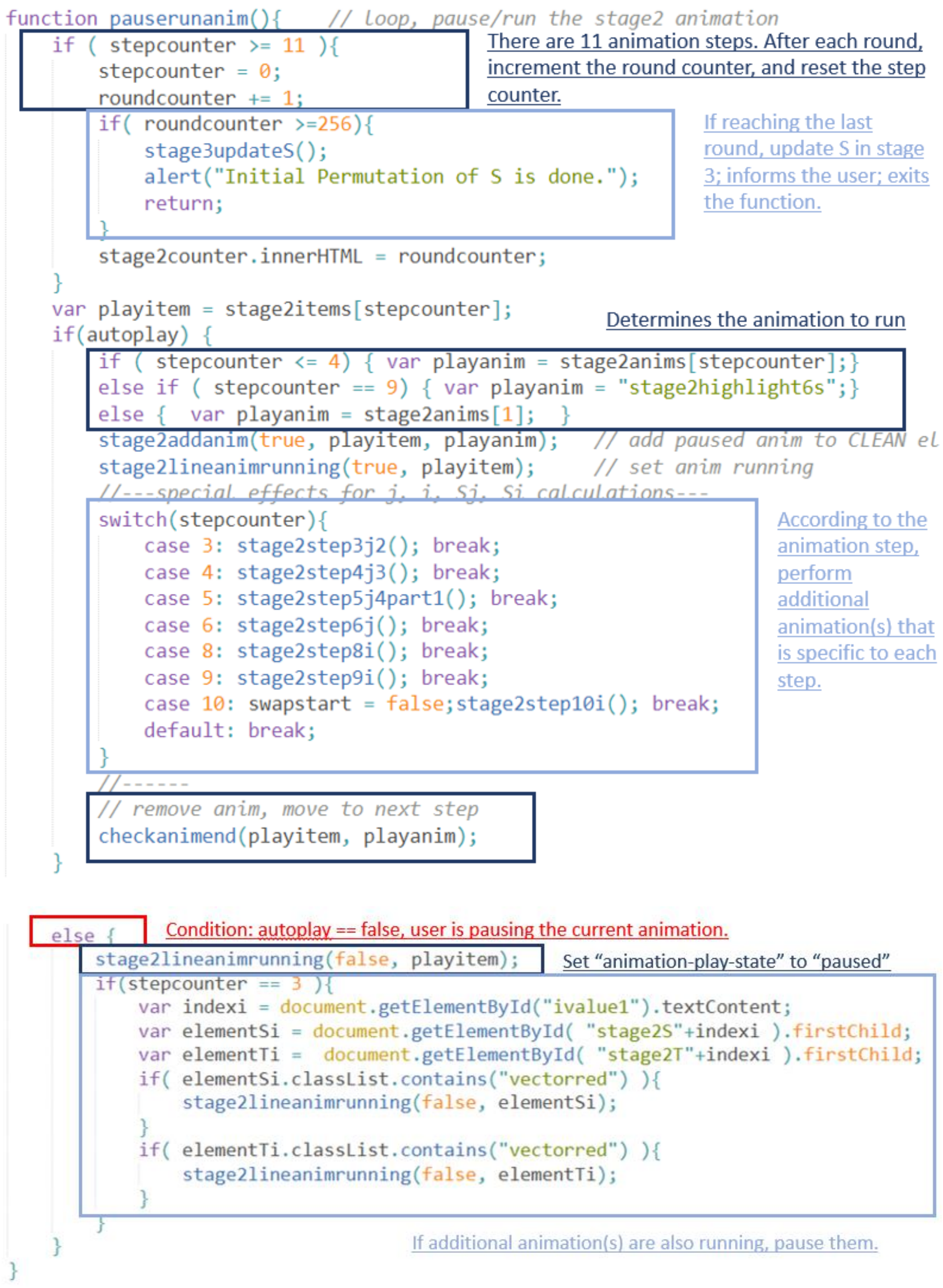
6. When the event is detected, the EventListener would call stage2addanim(false, targetelement, animation).
7. stage2addanim(false, targetelement, animation) removes the animation class from the target element, in preparation for the next round. It then calls stepincre().



```
function stepincre(){ stepcounter = stepcounter + 1; pauserunanim(); }
```

8. stepincre() increments stepcounter as the current animation step is at its end and the demo should move on to the next step. It then calls pauserunanim(), forming a closed circle.

Figure 21 explains the JavaScript implementation of pauserunanim():



```
function pauserunanim(){    // loop, pause/run the stage2 animation
    if ( stepcounter >= 11 ){
        stepcounter = 0;
        roundcounter += 1;
        if( roundcounter >=256){
            stage3updateS();
            alert("Initial Permutation of S is done.");
            return;
        }
        stage2counter.innerHTML = roundcounter;
    }
    var playitem = stage2items[stepcounter];
    if(autoplay) {
        if ( stepcounter <= 4) { var playanim = stage2anims[stepcounter];}
        else if ( stepcounter == 9) { var playanim = "stage2highlight6s";}
        else {  var playanim = stage2anims[1];   }
        stage2addanim(true, playitem, playanim);   // add paused anim to CLEAN el
        stage2lineanimrunning(true, playitem);     // set anim running
        //---special effects for j, i, Sj, Si calculations---
        switch(stepcounter){
            case 3: stage2step3j2(); break;
            case 4: stage2step4j3(); break;
            case 5: stage2step5j4part1(); break;
            case 6: stage2step6j(); break;
            case 8: stage2step8i(); break;
            case 9: stage2step9i(); break;
            case 10: swapstart = false;stage2step10i(); break;
            default: break;
        }
        //------
        // remove anim, move to next step
        checkanimend(playitem, playanim);
    }

    else {
        stage2lineanimrunning(false, playitem);
        if(stepcounter == 3 ){
            var indexi = document.getElementById("ivalue1").textContent;
            var elementSi = document.getElementById( "stage2S"+indexi ).firstChild;
            var elementTi =  document.getElementById( "stage2T"+indexi ).firstChild;
            if( elementSi.classList.contains("vectorred") ){
                stage2lineanimrunning(false, elementSi);
            }
            if( elementTi.classList.contains("vectorred") ){
                stage2lineanimrunning(false, elementTi);
            }
        }
    }
}
```

There are 11 animation steps. After each round, increment the round counter, and reset the step counter.

If reaching the last round, update S in stage 3; informs the user; exits the function.

Determines the animation to run

According to the animation step, perform additional animation(s) that is specific to each step.

Condition: autoplay == false, user is pausing the current animation.

Set "animation-play-state" to "paused"

If additional animation(s) are also running, pause them.

Figure 21. Code of pauserunanim() and comments

The real implementation of pauserunanim() is more complicated than the flow shown in Figure 20, because pauserunanim() is not only responsible of sequencing the animations, but also handles the pausing of animations and tracking of the number of rounds. This is summarized in Figure 22:

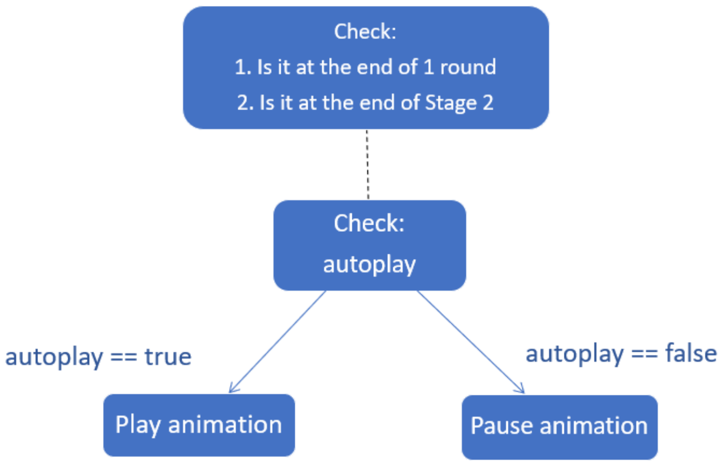
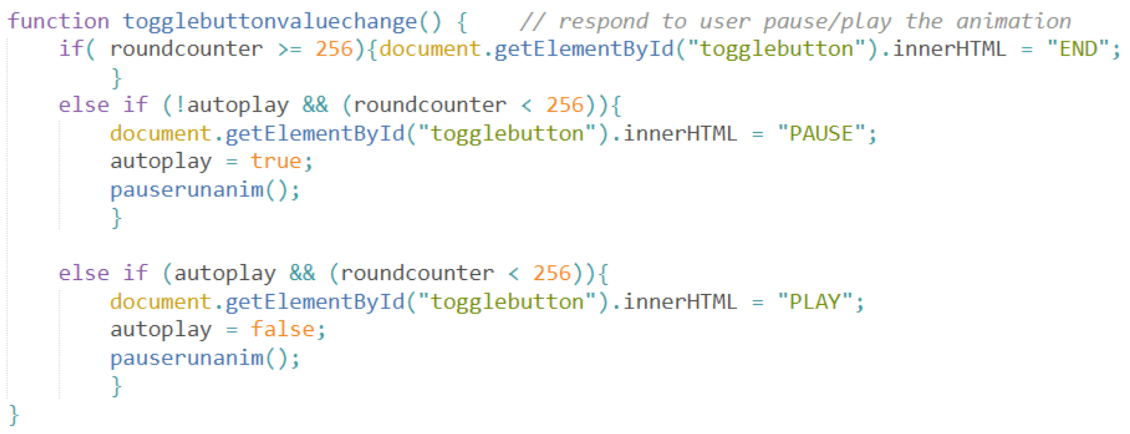


Figure 22. A very general flow of pauserunanim()

The value of the important variable autoplay is set by function togglebuttonvaluechange():



```
function togglebuttonvaluechange() {     // respond to user pause/play the animation
    if( roundcounter >= 256){document.getElementById("togglebutton").innerHTML = "END";
        }
    else if (!autoplay && (roundcounter < 256)){
        document.getElementById("togglebutton").innerHTML = "PAUSE";
        autoplay = true;
        pauserunanim();
        }

    else if (autoplay && (roundcounter < 256)){
        document.getElementById("togglebutton").innerHTML = "PLAY";
        autoplay = false;
        pauserunanim();
        }
}
```

Figure 23. Code of function togglebuttonvaluechange()

togglebuttonvaluechange() is only called when the user presses the "PLAY"/"PAUSE" button in Stage 2 as shown in Figure 24:

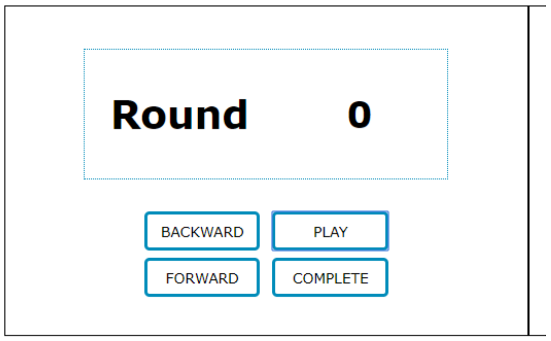


Figure 24. Control panel in web demo Stage 2

Combined, the two functions togglebuttonvaluechange() and pauserunanim() are able to:

1. Implement toggling of animations on and off
2. Implement infinite iterations of a series of animations
3. Implement a smooth animation sequence

They are at the core of the sequential web animation implementation discussed in this chapter.

3.3 Swap Animation Implementation

In the web demo, the most complex animation step is the step where two S elements swap positions. Take the swapping of S[5] and S[11] in Stage 3 as an example:

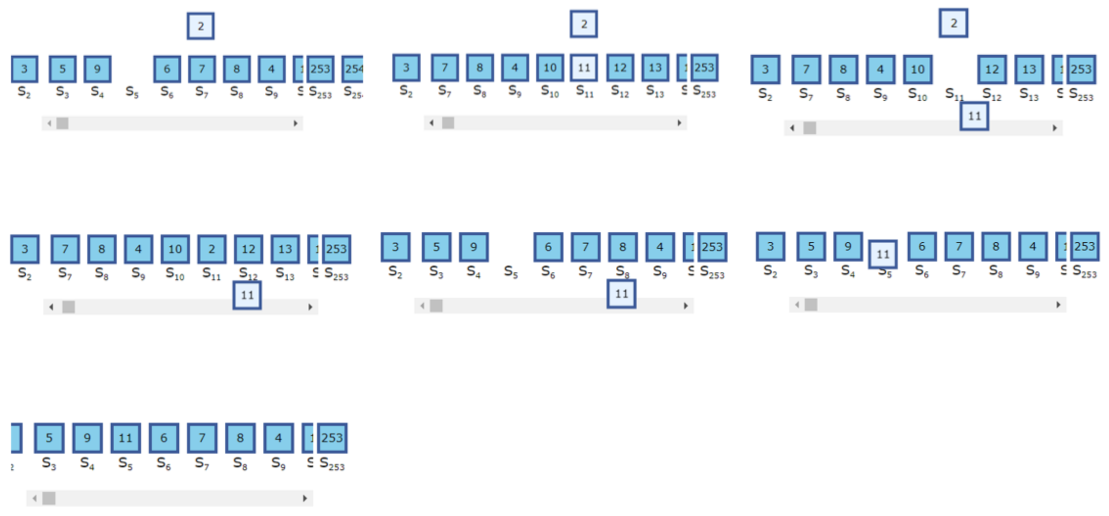


Figure 25. screenshots taken in the swap process of S[5] and S[11] in Stage 3

Each swap step involves a few sub-steps. Each sub-step is pausable. For instance, in the screenshot located in the center of Figure 25, value of S[11] is halfway to the position of S[5]. There is no built-in function that can realize animation of two HTML elements move and swap positions with each other as such. Therefore, in the web demo, swap animation is built up using many basic animation methods in JavaScript and jQuery.

Section 3.3.1 explains those basic animation methods. Section 3.3.2 explains the flow of basic animations in an abstract way. Section 3.3.3 explains the implementation code in details and concludes this chapter.

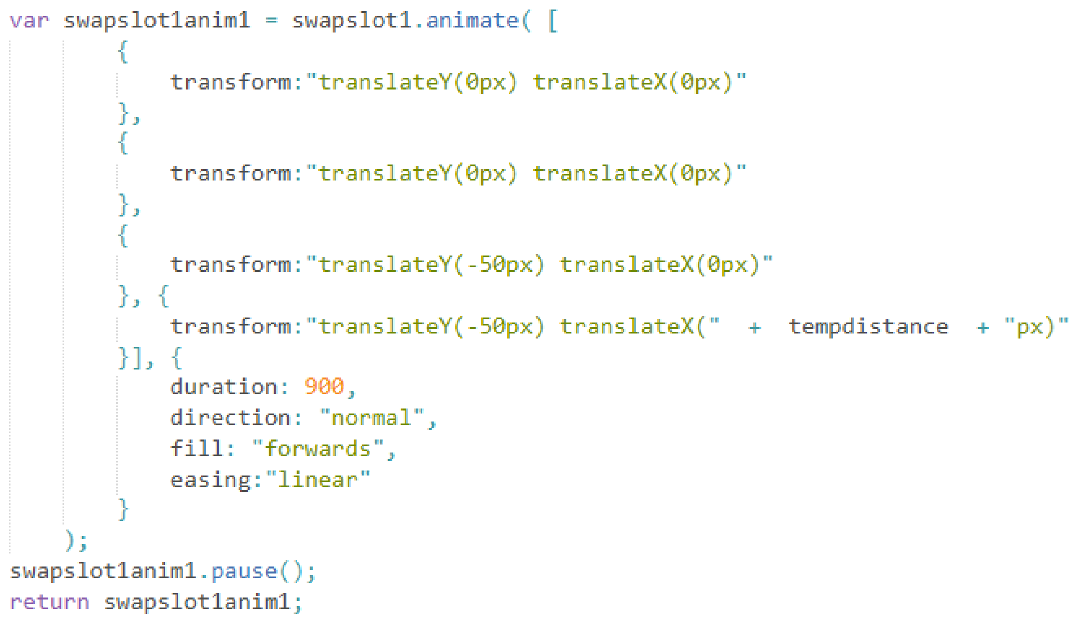
3.3.1 Basic Animation Methods in JavaScript and jQuery

3.3.1.1 JavaScript animation

In Sub-chapter 3.2, JavaScript is used to empower CSS animations. In fact, JavaScript alone can create diverse animation effects through a web API: Element.animate() [19].

Syntax: var animation = element.animate(keyframes, options);

Keyframes are similar with CSS Keyframes in which the phases of animation are described in animatable CSS properties. Options allow the user customize animation duration, delay duration (default is 0), animation direction, whether the animation effects are retained after animation ends and so on. Figure 26. is an example:

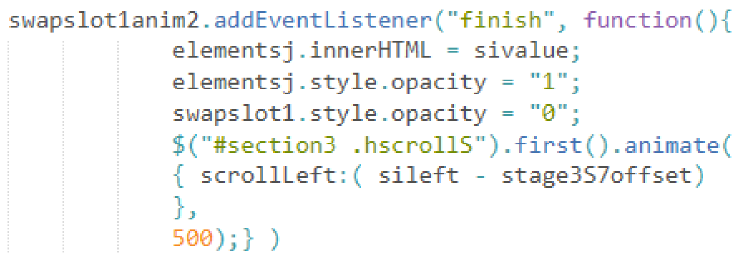


```
var swapslot1anim1 = swapslot1.animate( [
            {
                transform:"translateY(0px) translateX(0px)"
            },
            {
                transform:"translateY(0px) translateX(0px)"
            },
            {
                transform:"translateY(-50px) translateX(0px)"
            }, {
                transform:"translateY(-50px) translateX("  +  tempdistance  + "px)"
            }], {
                duration: 900,
                direction: "normal",
                fill: "forwards",
                easing:"linear"
            }
    );
swapslot1anim1.pause();
return swapslot1anim1;
```

Figure 26. element swapslot1.animate() details

In Figure 26, “fill” determines whether the animation effects are retained after animation ends. “forwards” means the effects are retained. However, a hidden fact about this option is that the retained animation effects would be removed if new animations are set running on the same element. This hidden aspect of “forward” option caused bizarre bugs during the web demo implementations, when one element underwent a chain of animations in which every animation built on its predecessor. Another option, “easing”, determines the rate of animation’s change over time.

This web API also allows the user to toggle the animation on and off by calling pause() / play() on the variable animation. This variable animation serves as a reference to the animation effect(s), and can be returned by a function. Moreover, it has its own version of “animationend” – “finish”.

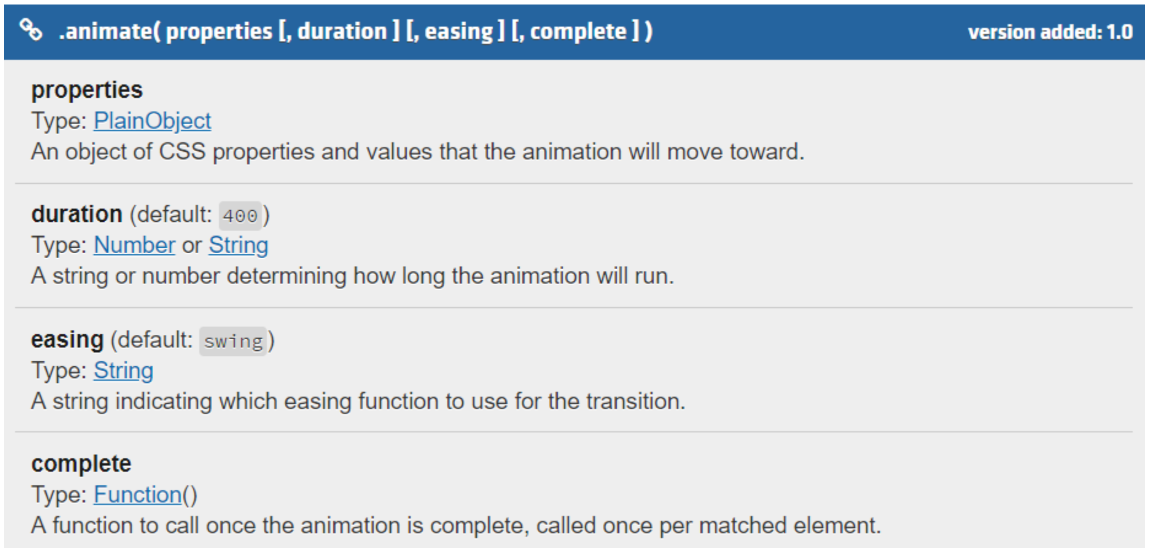


```
swapslot1anim2.addEventListener("finish", function(){
            elementsj.innerHTML = sivalue;
            elementsj.style.opacity = "1";
            swapslot1.style.opacity = "0";
            $("#section3 .hscrollS").first().animate(
            { scrollLeft:( sileft - stage3S7offset)
            },
            500);} )
```

Figure 27. EventListener on an animation object referred to by “swapslot1anim2”

The capability of keeping track of the completion of JavaScript animations is significant because it allows the sequential and pausable web animation techniques described in Sub-Chapter 3.2 to be utilized in swap animation implementation.

3.3.1.2 jQuery animation



.animate(properties [, duration] [, easing] [, complete])   version added: 1.0

properties
Type: PlainObject
An object of CSS properties and values that the animation will move toward.

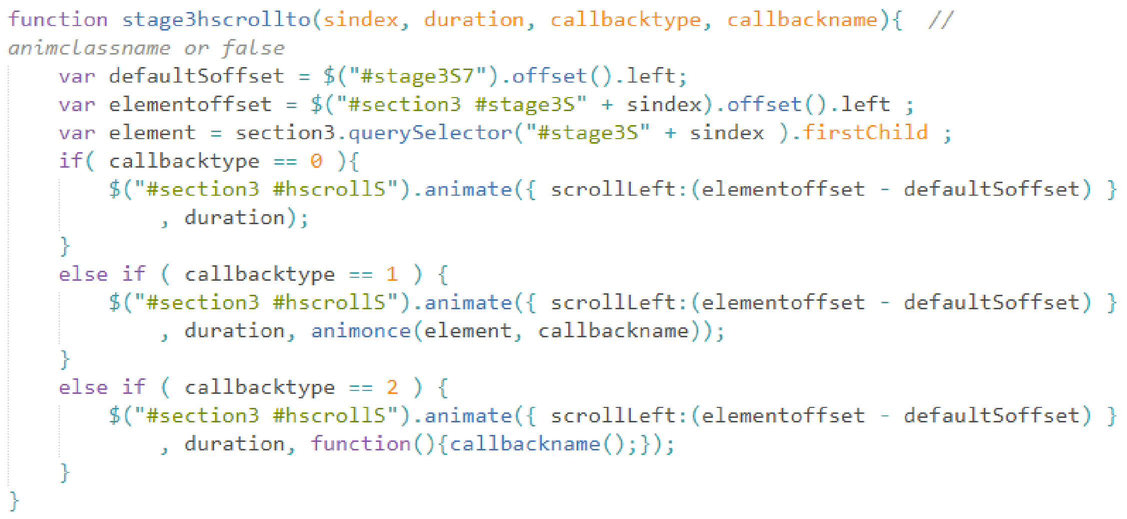
duration (default: `400`)
Type: Number or String
A string or number determining how long the animation will run.

easing (default: `swing`)
Type: String
A string indicating which easing function to use for the transition.

complete
Type: Function()
A function to call once the animation is complete, called once per matched element.

Figure 28. Screenshot of online jQuery documentation on jQuery animate() method [20]. Refer to link in the references for more details

The effects of this jQuery animation method are best illustrated with an example such as this function stage3hscrollto(sindex, duration, callbacktype, callbackname) in Figure 29.



```
function stage3hscrollto(sindex, duration, callbacktype, callbackname){  //
animclassname or false
    var defaultSoffset = $("#stage3S7").offset().left;
    var elementoffset = $("#section3 #stage3S" + sindex).offset().left ;
    var element = section3.querySelector("#stage3S" + sindex ).firstChild ;
    if( callbacktype == 0 ){
        $("#section3 #hscrollS").animate({ scrollLeft:(elementoffset - defaultSoffset) }
            , duration);
    }
    else if ( callbacktype == 1 ) {
        $("#section3 #hscrollS").animate({ scrollLeft:(elementoffset - defaultSoffset) }
            , duration, animonce(element, callbackname));
    }
    else if ( callbacktype == 2 ) {
        $("#section3 #hscrollS").animate({ scrollLeft:(elementoffset - defaultSoffset) }
            , duration, function(){callbackname();});
    }
}
```

Figure 29. Stage 3 scrolling function which uses jQuery animate()

This function works on the scroll element in Stage 3 as shown in Figure 30:

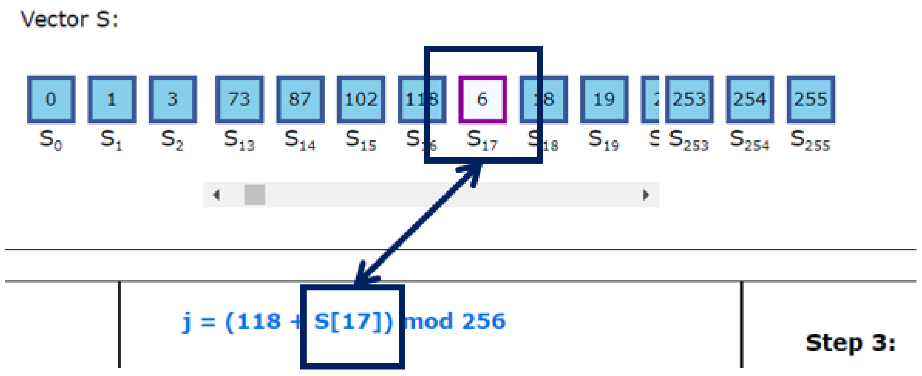
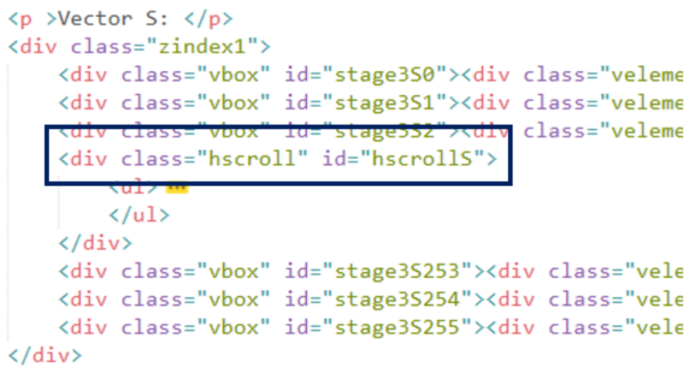


Figure 30. Scrolled to S[17] and changed its color scheme

$("#section3 #hscrollS") accesses the HTML scroll element for the function. "scrollLeft" defines the animation to perform, and takes in a value calculated by (elementoffset – defaultSoffset).



```
<p >Vector S: </p>
<div class="zindex1">
    <div class="vbox" id="stage3S0"><div class="veleme
    <div class="vbox" id="stage3S1"><div class="veleme
    <div class="vbox" id="stage3S2"><div class="veleme
    <div class="hscroll" id="hscrollS">
        <ul> ...
        </ul>
    </div>
    <div class="vbox" id="stage3S253"><div class="vele
    <div class="vbox" id="stage3S254"><div class="vele
    <div class="vbox" id="stage3S255"><div class="vele
</div>
```

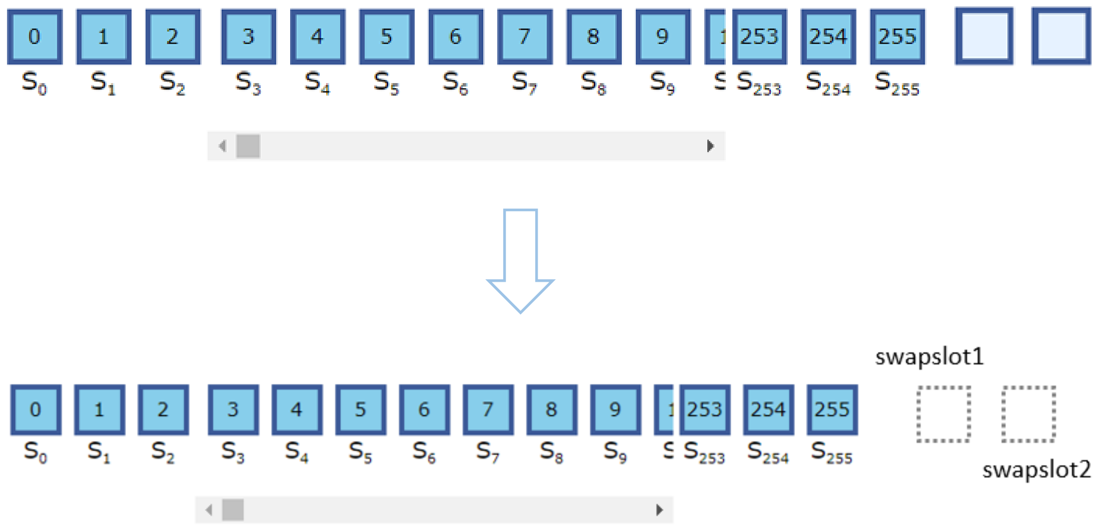
Figure 31. scroll element in Stage 3

The change of color scheme is caused by passing integer 1 as the callbacktype parameter. It causes the animate() function to have a callback function named animonce(element, callbackname). After the scrolling animation completes, the target S element is in view of the user and animonce(element, callbackname) is fired to set a color-changing animation on the S element.

In summary, the jQuery animation method offers many options and jQuery-custom properties such as "scrollLeft", "scrollTop". One unique option is the possibility of callback function. It is slightly different from the JavaScript animation method in implementation details. Besides, unlike Element.animate() API, it does not return an object.

3.3.2 Abstract Flow of Swap Animation

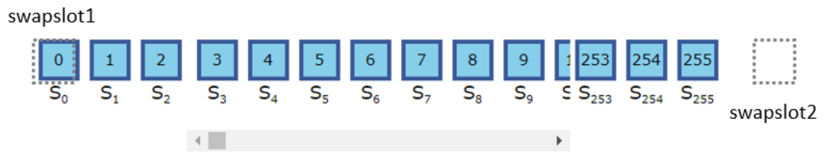
In the HTML document, two empty vector elements are created and positioned besides vector S. Their opacities are set to 0. These two invisible elements as named swapslot1 and swapslot2.



Below is the flow of swap animation:

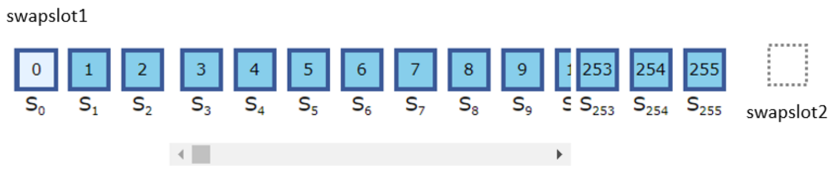
Step 1. Scroll vector S so that S[i] is visible in the browser window.

Step 2. Move swapslot1 to the position of S[i] (S[0] is used as an example in the illustration).

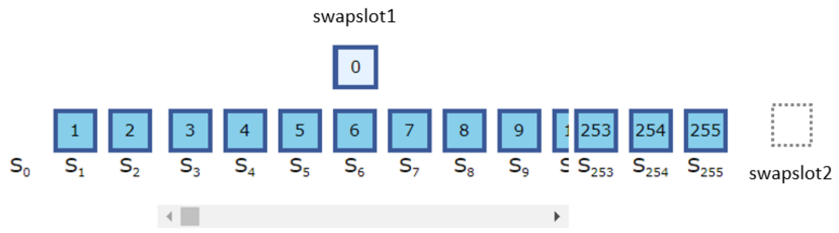

swapslot1
0
1
2
3
4
5
6
7
8
9
253
254
255
S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{253} S_{254} S_{255}
swapslot2

Step 3. swapslot1 copies the value of S[i].

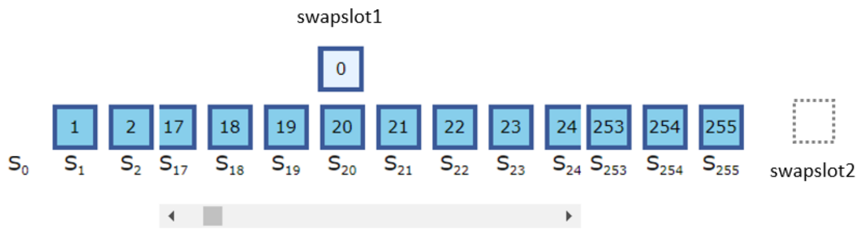
Step 4. Set opacity of swapslot1 to 1 and opacity of S[i] to 0.


swapslot1
0
1
2
3
4
5
6
7
8
9
253
254
255
S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{253} S_{254} S_{255}
swapslot2

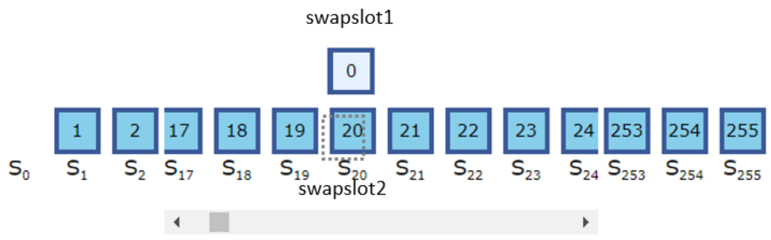
Step 5. Moves swapslot1 to the center of and above the scroll bar.


swapslot1
0
1
2
3
4
5
6
7
8
9
253
254
255
S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{253} S_{254} S_{255}
swapslot2

Step 6. Scroll to S[j]. S[j] is now at the center of the scroll bar. (S[20] is used as an example in the illustration). S[j] may be at the beginning or the end of the vector S if j is less than 7 or greater than 249.

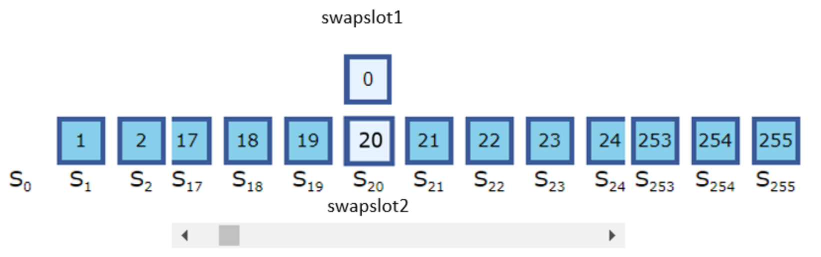

swapslot1
0
1
2
17
18
19
20
21
22
23
24
253
254
255
S_0 S_1 S_2 S_{17} S_{18} S_{19} S_{20} S_{21} S_{22} S_{23} S_{24} S_{253} S_{254} S_{255}
swapslot2

Step 7. Move swapslot2 to the position of S[j].

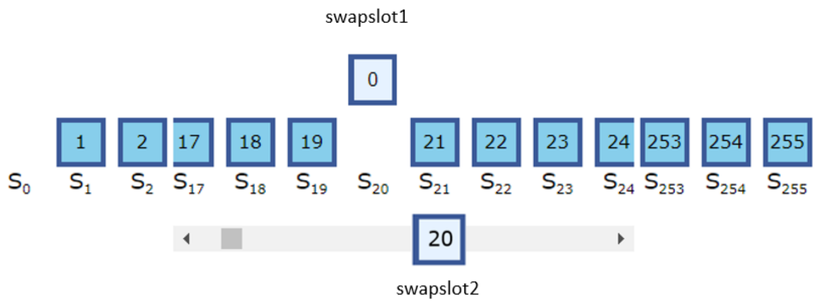

swapslot1
0
1
2
17
18
19
20
21
22
23
24
253
254
255
S_0 S_1 S_2 S_{17} S_{18} S_{19} S_{20} S_{21} S_{22} S_{23} S_{24} S_{253} S_{254} S_{255}
swapslot2

Step 8. swapslot2 copies the value of S[j].

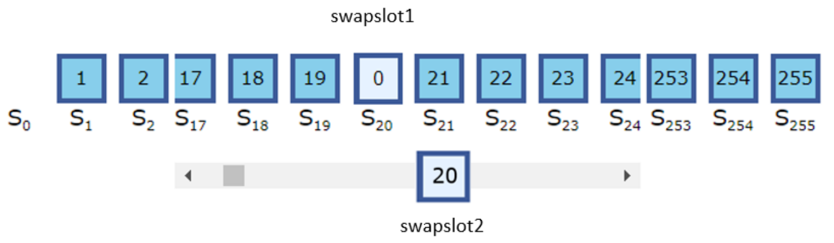
Step 9. Set opacity of swapslot2 to 1 and opacity of S[j] to 0.



Step 10. Move swapslot2 below the vector S and to one side.

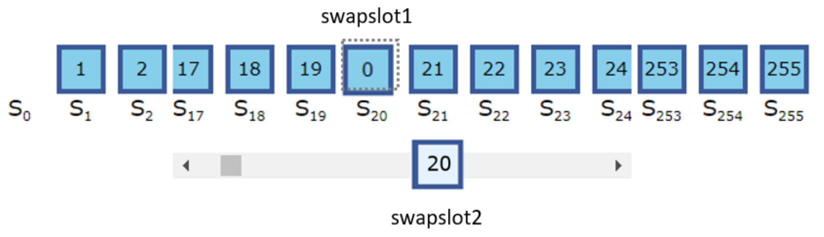


Step 11. swapslot1 moves down to fill in the blank above “S_{20}”.

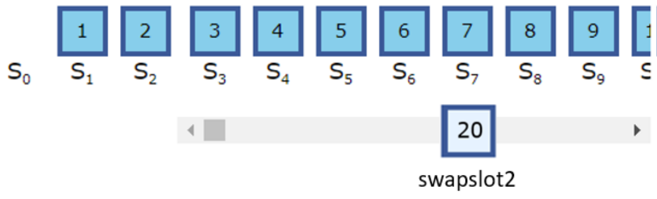


Step 12. S[j] copies the value of swapslot1.

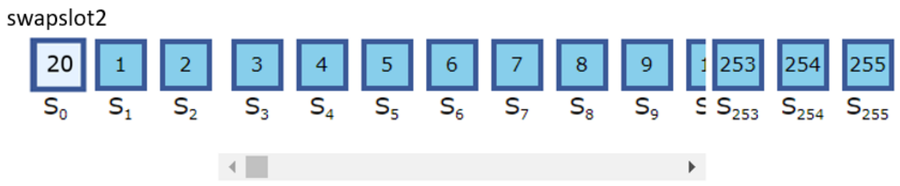
Step 13. Set opacity of S[j] to 1, set opacity of swapslot1 to 0.



Step 14. Scroll to S[i]. S[i] is now at the center of the scroll bar if possible. S[i] may be at the beginning or the end of the vector S if i is less than 7 or greater than 249.

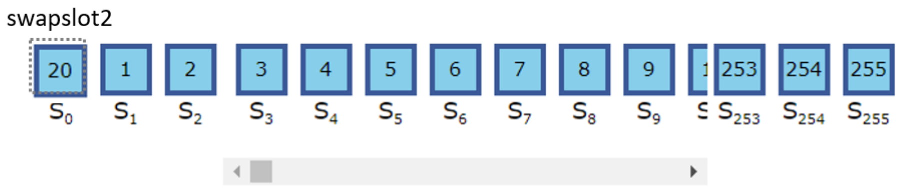

1 2 3 4 5 6 7 8 9 253 254 255
S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{253} S_{254} S_{255}
20
swapslot2

Step 15. swapslot2 moves down to fill in the blank above “S_0”.


swapslot2
20 1 2 3 4 5 6 7 8 9 253 254 255
S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{253} S_{254} S_{255}

Step 16. S[i] copies the value of swapslot2.

Step 17. Set opacity of S[i] to 1, set opacity of swapslot2 to 0.


swapslot2
20 1 2 3 4 5 6 7 8 9 253 254 255
S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{253} S_{254} S_{255}

End of swap animation.

The animated swapping process is not performed by the S[i] and S[j] elements as seen in the RC4 algorithm, but by the two elements named swapslot1 and swapslot2. In fact, the S[i] and S[j] elements swapped values in an instant as well as in an unanimated way. They appear to be swapped only because of the opacity changes. This counterintuitive design is chosen because it has been found to be the most bug-free one after numerous experiments.

One experiment found that lifting an element out of and above the scroll bar in which it originally belonged to, would cause the element to ‘disappear’ because the HTML element above the scroll bar occupied a certain amount of space and concealed the lifted element. Another experiment found that lifting the square element out of the vector would result in the effect shown in Figure 32:

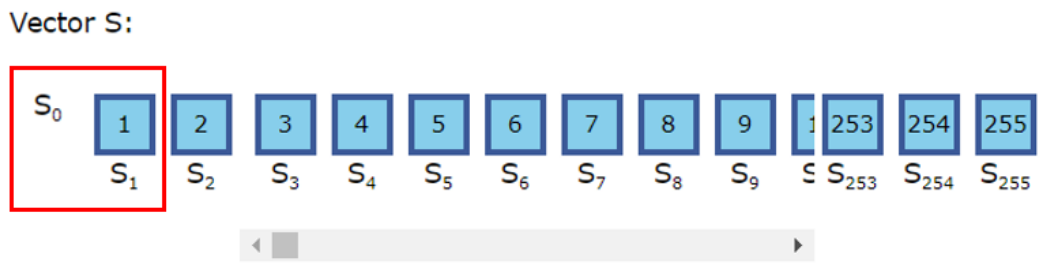
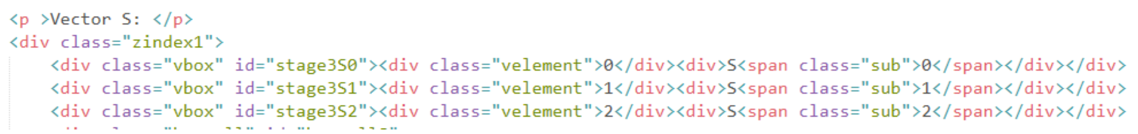

Vector S:
S_0
1 2 3 4 5 6 7 8 9 253 254 255
S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{253} S_{254} S_{255}

Figure 32. undesirable effect of lifting value of S[0] out of the vector S

In older design, JavaScript and jQuery accessed the S elements with their unique IDs. These ID had the same values as the S indexes shown under every element. Swapping the elements led to swapping of the IDs too. After a few rounds of stream generation, the S indexes and the element IDs diverged greatly. Finding the right S[i] and S[j] elements to be swapped was still possible, but very difficult and cumbersome to code.

This issue eventually leads to separation of the elements and the reference IDs. In the current design, the HTML of vector S is as shown below:

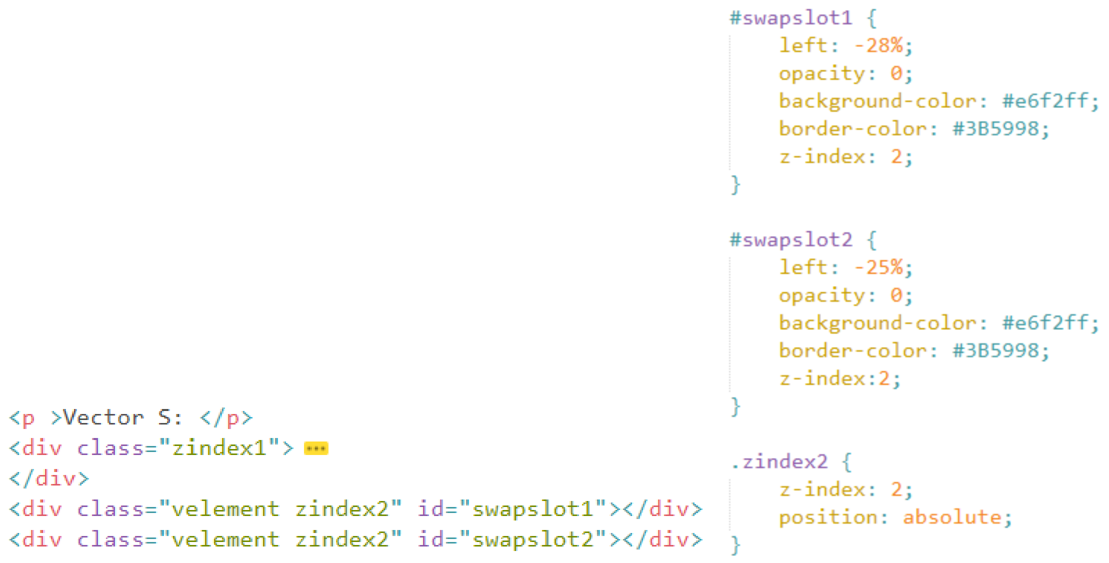


```
<p >Vector S: </p>
<div class="zindex1">
    <div class="vbox" id="stage3S0"><div class="velement">0</div><div>S<span class="sub">0</span></div></div>
    <div class="vbox" id="stage3S1"><div class="velement">1</div><div>S<span class="sub">1</span></div></div>
    <div class="vbox" id="stage3S2"><div class="velement">2</div><div>S<span class="sub">2</span></div></div>
```

The IDs now belong to the parent element -- <div class="vbox" id="stage3S[index]">. However, this has another side effect: the S elements, once lifted out of their parent elements, are indistinguishable. Implementation of two sets of distinct IDs were contemplated, but the current implementation was chosen instead for its simplicity.

3.3.3 Explanation of Code and Implementation Details

JavaScript code is explained, and some significant details are mentioned in this section.

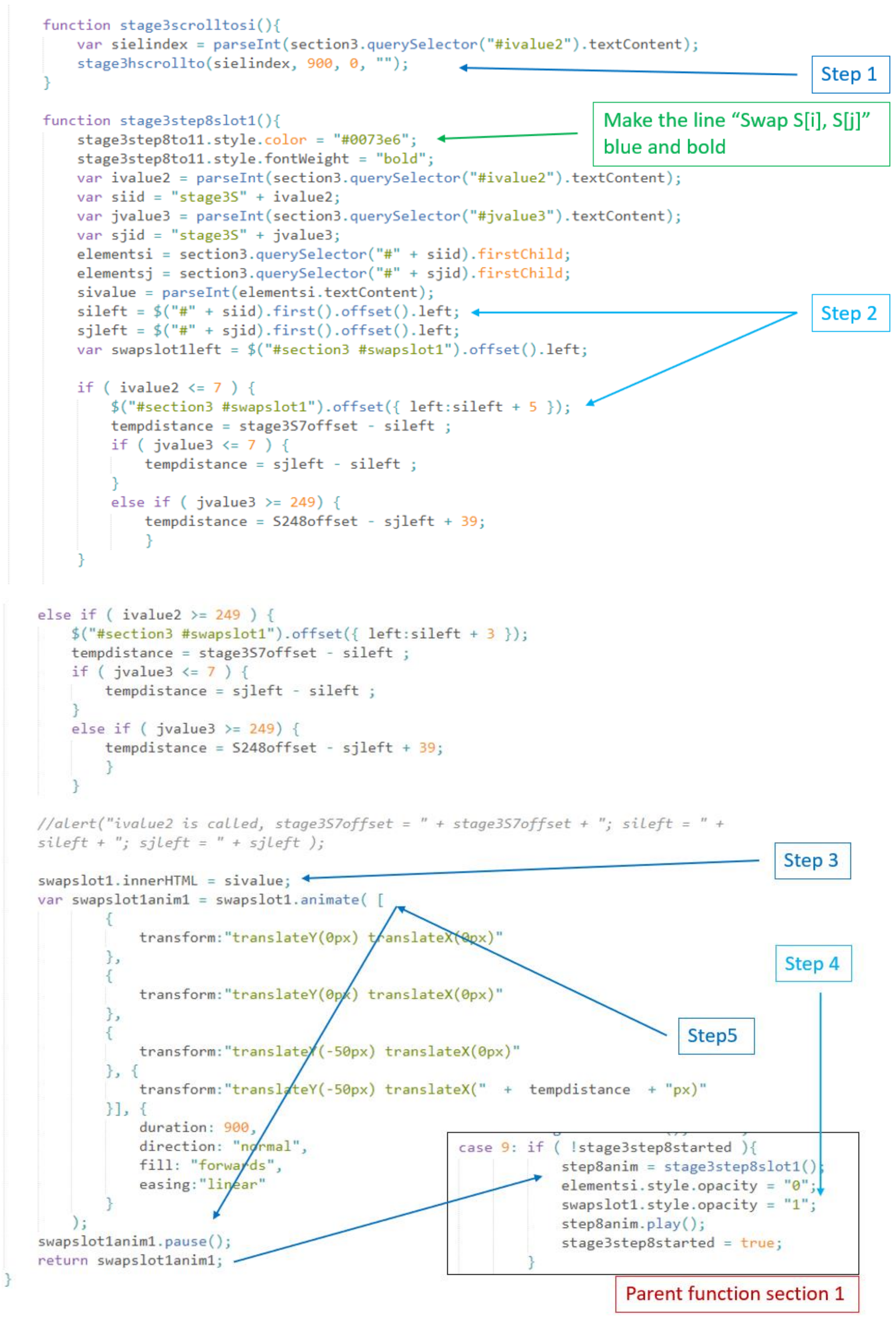


```
<p >Vector S: </p>
<div class="zindex1"> ...
</div>
<div class="velement zindex2" id="swapslot1"></div>
<div class="velement zindex2" id="swapslot2"></div>
```

```
#swapslot1 {
    left: -28%;
    opacity: 0;
    background-color: #e6f2ff;
    border-color: #3B5998;
    z-index: 2;
}

#swapslot2 {
    left: -25%;
    opacity: 0;
    background-color: #e6f2ff;
    border-color: #3B5998;
    z-index:2;
}

.zindex2 {
    z-index: 2;
    position: absolute;
}
```

Figure 33. HTML & CSS of the two elements swapslot1 and swapslot2

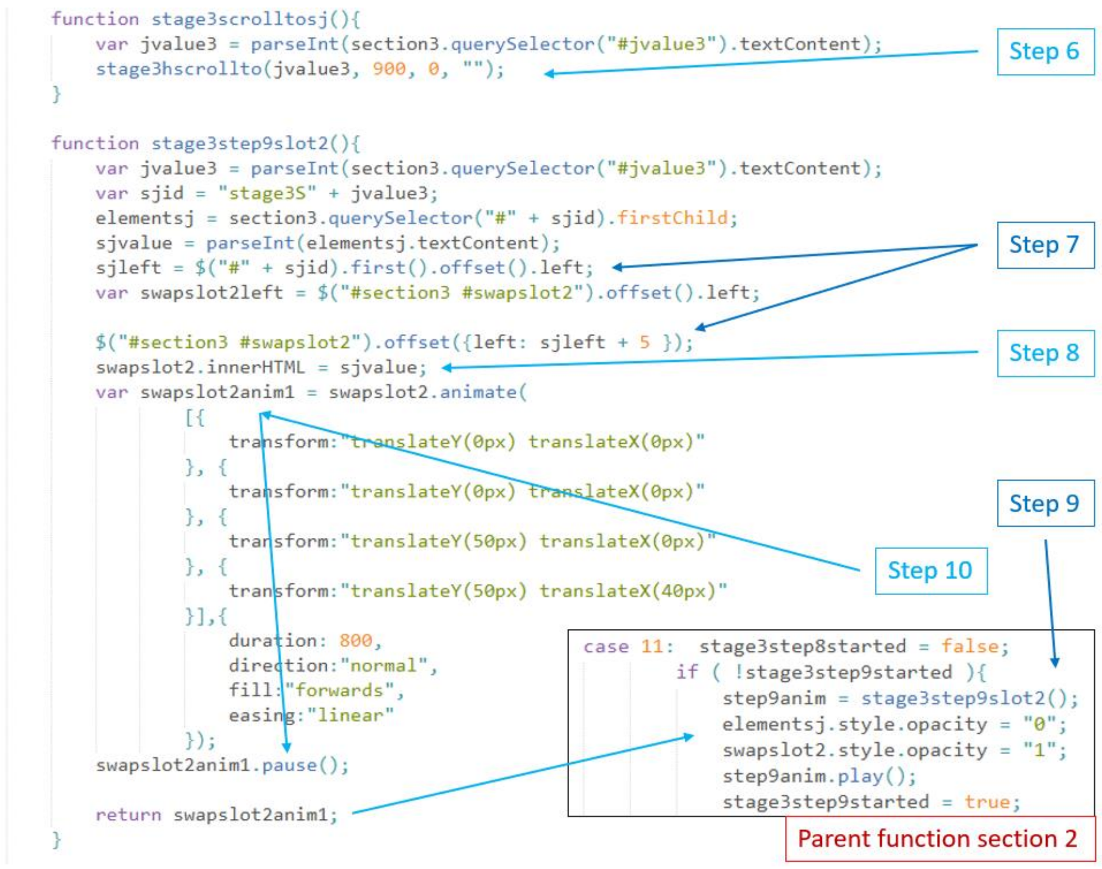
Code section below shows functions and their corresponding steps described in 3.3.2 Abstract Flow of Swap Animation.



```
function stage3scrolltosi(){
    var sielindex = parseInt(section3.querySelector("#ivalue2").textContent);
    stage3hscrollto(sielindex, 900, 0, "");
}

function stage3step8slot1(){
    stage3step8to11.style.color = "#0073e6";
    stage3step8to11.style.fontWeight = "bold";
    var ivalue2 = parseInt(section3.querySelector("#ivalue2").textContent);
    var siid = "stage3S" + ivalue2;
    var jvalue3 = parseInt(section3.querySelector("#jvalue3").textContent);
    var sjid = "stage3S" + jvalue3;
    elementsi = section3.querySelector("#" + siid).firstChild;
    elementsj = section3.querySelector("#" + sjid).firstChild;
    sivalue = parseInt(elementsi.textContent);
    sileft = $("#" + siid).first().offset().left;
    sjleft = $("#" + sjid).first().offset().left;
    var swapslot1left = $("#section3 #swapslot1").offset().left;

    if ( ivalue2 <= 7 ) {
        $("#section3 #swapslot1").offset({ left:sileft + 5 });
        tempdistance = stage3S7offset - sileft ;
        if ( jvalue3 <= 7 ) {
            tempdistance = sjleft - sileft ;
        }
        else if ( jvalue3 >= 249) {
            tempdistance = S248offset - sjleft + 39;
            }
    }

    else if ( ivalue2 >= 249 ) {
        $("#section3 #swapslot1").offset({ left:sileft + 3 });
        tempdistance = stage3S7offset - sileft ;
        if ( jvalue3 <= 7 ) {
            tempdistance = sjleft - sileft ;
        }
        else if ( jvalue3 >= 249) {
            tempdistance = S248offset - sjleft + 39;
            }
        }

    //alert("ivalue2 is called, stage3S7offset = " + stage3S7offset + "; sileft = " +
    sileft + "; sjleft = " + sjleft );

    swapslot1.innerHTML = sivalue;
    var swapslot1anim1 = swapslot1.animate( [
            {
                transform:"translateY(0px) translateX(0px)"
            },
            {
                transform:"translateY(0px) translateX(0px)"
            },
            {
                transform:"translateY(-50px) translateX(0px)"
            }, {
                transform:"translateY(-50px) translateX(" + tempdistance + "px)"
            }], {
                duration: 900,
                direction: "normal",
                fill: "forwards",
                easing:"linear"
            }
        );
    swapslot1anim1.pause();
    return swapslot1anim1;
}
```

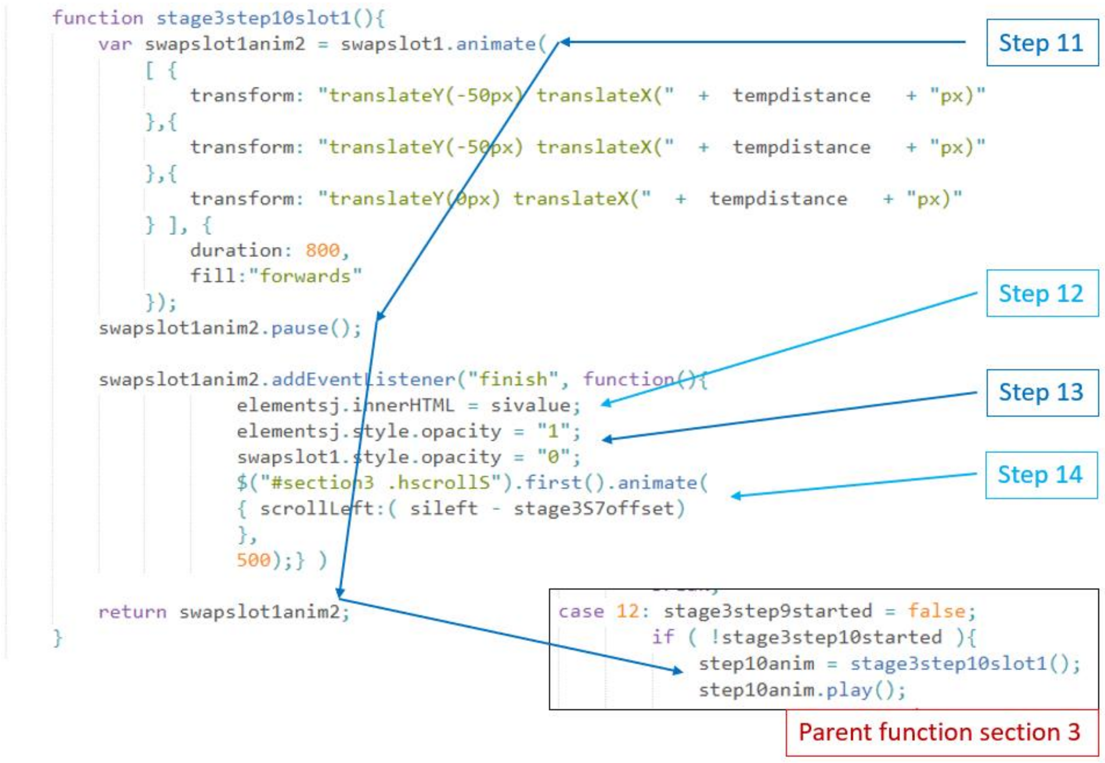
```
case 9: if ( !stage3step8started ){
            step8anim = stage3step8slot1();
            elementsi.style.opacity = "0";
            swapslot1.style.opacity = "1";
            step8anim.play();
            stage3step8started = true;
        }
```

Step 1
Make the line “Swap S[i], S[j]” blue and bold
Step 2
Step 3
Step 4
Step5
Parent function section 1



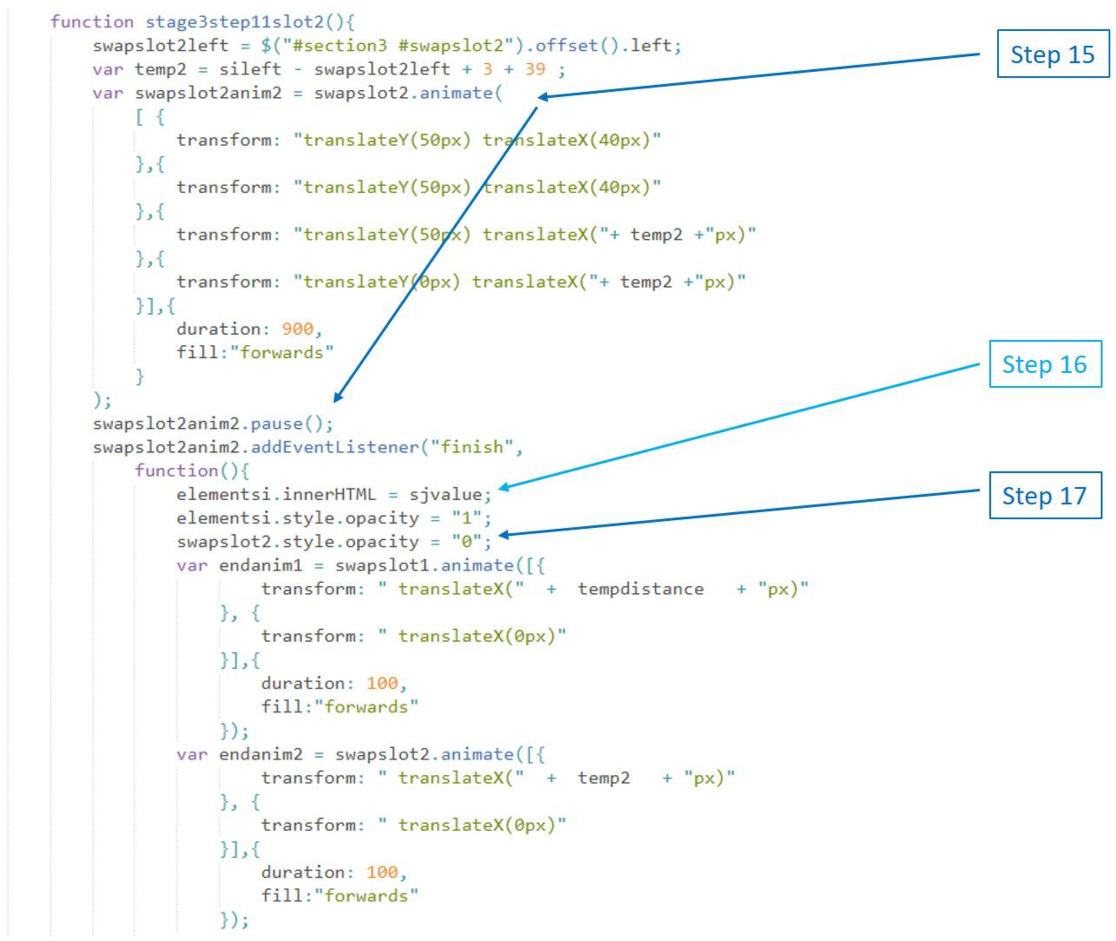
```
function stage3scrolltosj(){
    var jvalue3 = parseInt(section3.querySelector("#jvalue3").textContent);
    stage3hscrollto(jvalue3, 900, 0, "");
}

function stage3step9slot2(){
    var jvalue3 = parseInt(section3.querySelector("#jvalue3").textContent);
    var sjid = "stage3S" + jvalue3;
    elementsj = section3.querySelector("#" + sjid).firstChild;
    sjvalue = parseInt(elementsj.textContent);
    sjleft = $("#" + sjid).first().offset().left;
    var swapslot2left = $("#section3 #swapslot2").offset().left;

    $("#section3 #swapslot2").offset({left: sjleft + 5 });
    swapslot2.innerHTML = sjvalue;
    var swapslot2anim1 = swapslot2.animate(
            [{
                transform:"translateY(0px) translateX(0px)"
            }, {
                transform:"translateY(0px) translateX(0px)"
            }, {
                transform:"translateY(50px) translateX(0px)"
            }, {
                transform:"translateY(50px) translateX(40px)"
            }],{
                duration: 800,
                direction:"normal",
                fill:"forwards",
                easing:"linear"
            });
    swapslot2anim1.pause();

    return swapslot2anim1;
}
```

```
case 11:  stage3step8started = false;
        if ( !stage3step9started ){
            step9anim = stage3step9slot2();
            elementsj.style.opacity = "0";
            swapslot2.style.opacity = "1";
            step9anim.play();
            stage3step9started = true;
```



```
function stage3step10slot1(){
    var swapslot1anim2 = swapslot1.animate(
        [ {
            transform: "translateY(-50px) translateX("  +  tempdistance   + "px)"
        },{
            transform: "translateY(-50px) translateX("  +  tempdistance   + "px)"
        },{
            transform: "translateY(0px) translateX("  +  tempdistance   + "px)"
        } ], {
            duration: 800,
            fill:"forwards"
        });
    swapslot1anim2.pause();

    swapslot1anim2.addEventListener("finish", function(){
                elementsj.innerHTML = sivalue;
                elementsj.style.opacity = "1";
                swapslot1.style.opacity = "0";
                $("#section3 .hscrollS").first().animate(
                { scrollLeft:( sileft - stage3S7offset)
                },
                500);} )

    return swapslot1anim2;
}
```

```
case 12: stage3step9started = false;
        if ( !stage3step10started ){
            step10anim = stage3step10slot1();
            step10anim.play();
```

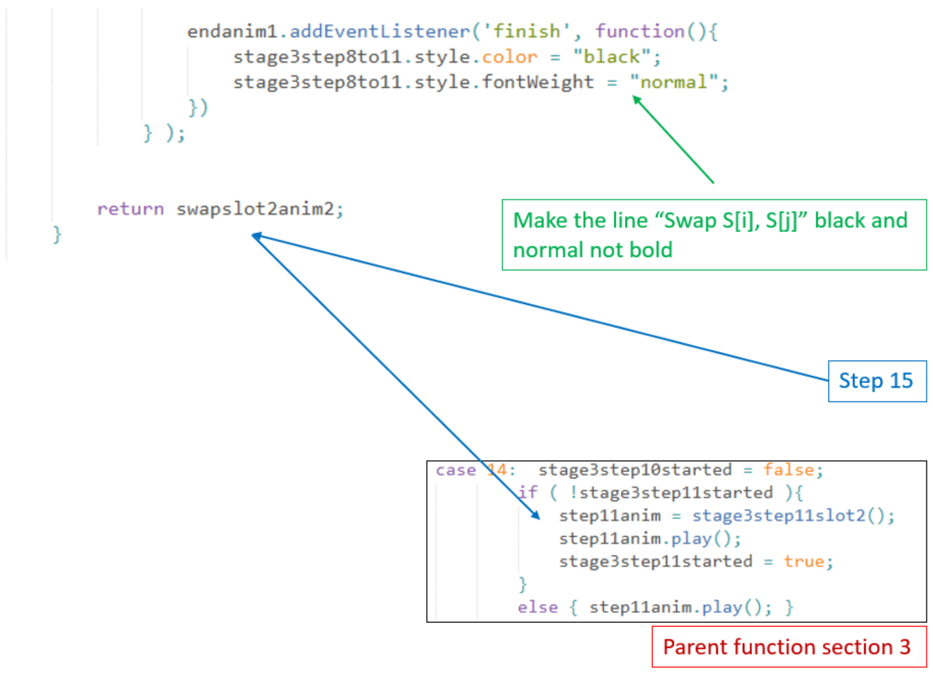


```
function stage3step11slot2(){
    swapslot2left = $("#section3 #swapslot2").offset().left;
    var temp2 = sileft - swapslot2left + 3 + 39 ;
    var swapslot2anim2 = swapslot2.animate(
        [ {
            transform: "translateY(50px) translateX(40px)"
        },{
            transform: "translateY(50px) translateX(40px)"
        },{
            transform: "translateY(50px) translateX("+ temp2 +"px)"
        },{
            transform: "translateY(0px) translateX("+ temp2 +"px)"
        }],{
            duration: 900,
            fill:"forwards"
        }
    );
    swapslot2anim2.pause();
    swapslot2anim2.addEventListener("finish",
        function(){
            elementsi.innerHTML = sjvalue;
            elementsi.style.opacity = "1";
            swapslot2.style.opacity = "0";
            var endanim1 = swapslot1.animate([{
                    transform: " translateX("  +  tempdistance   + "px)"
                }, {
                    transform: " translateX(0px)"
                }],{
                    duration: 100,
                    fill:"forwards"
                });
            var endanim2 = swapslot2.animate([{
                    transform: " translateX("  +  temp2   + "px)"
                }, {
                    transform: " translateX(0px)"
                }],{
                    duration: 100,
                    fill:"forwards"
                });
```

In the code section above, animations endanim1 and endanim2 cause elements swapslot1 and swapslot2 to be transformed a certain distance on x-axis. This is to solve a problem mentioned in the beginning of this chapter:

"However, a hidden fact about this option is that the retained animation effects would be removed if new animations are set running on the same element. This hidden aspect of "forward" option caused bizarre bugs during the web demo implementations, when one element underwent a chain of animations in which every animation built on its predecessor."

The two animations cancel out any x-axis transformation on swapslot1 and swapslot2 at the end of the swap animation step. In the next round, new animations set on swapslot1 and swapslot2 would run correctly.

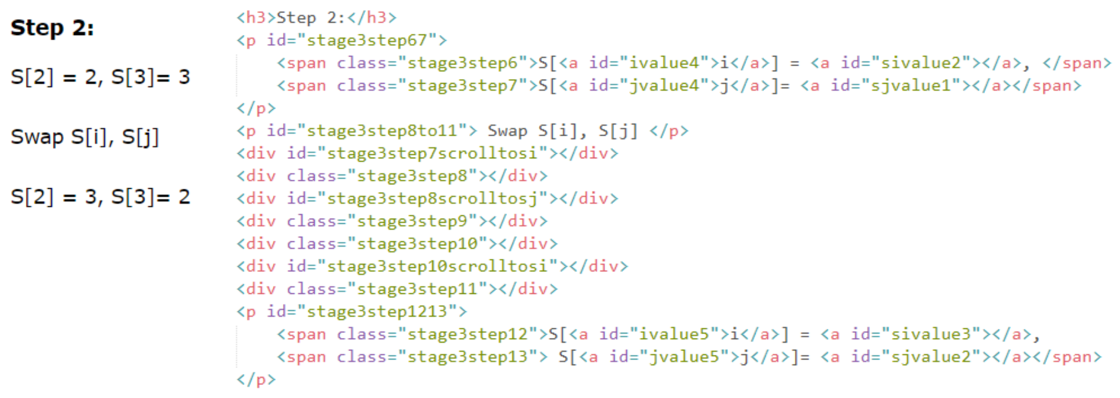


```
            endanim1.addEventListener('finish', function(){
                stage3step8to11.style.color = "black";
                stage3step8to11.style.fontWeight = "normal";
            })
        } );

    return swapslot2anim2;
}
```

Make the line “Swap S[i], S[j]” black and normal not bold
Step 15
Parent function section 3

```
case 14:  stage3step10started = false;
          if ( !stage3step11started ){
              step11anim = stage3step11slot2();
              step11anim.play();
              stage3step11started = true;
          }
          else { step11anim.play(); }
```

Below is a comparison of what is displayed in the webpage and what is written in the HTML document.



Step 2:

S[2] = 2, S[3]= 3

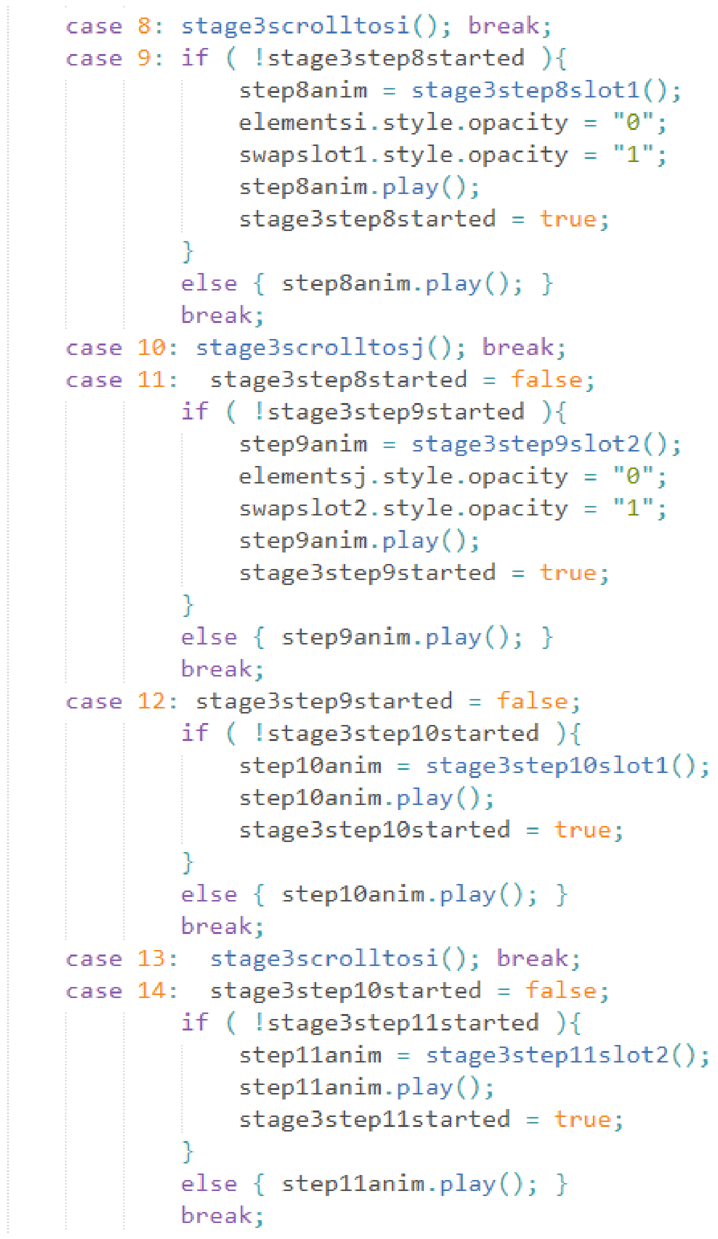
Swap S[i], S[j]

S[2] = 3, S[3]= 2

```
<h3>Step 2:</h3>
<p id="stage3step67">
    <span class="stage3step6">S[<a id="ivalue4">i</a>] = <a id="sivalue2"></a>, </span>
    <span class="stage3step7">S[<a id="jvalue4">j</a>]= <a id="sjvalue1"></a></span>
</p>
<p id="stage3step8to11"> Swap S[i], S[j] </p>
<div id="stage3step7scrolltosi"></div>
<div class="stage3step8"></div>
<div id="stage3step8scrolltosj"></div>
<div class="stage3step9"></div>
<div class="stage3step10"></div>
<div id="stage3step10scrolltosi"></div>
<div class="stage3step11"></div>
<p id="stage3step1213">
    <span class="stage3step12">S[<a id="ivalue5">i</a>] = <a id="sivalue3"></a>,
    <span class="stage3step13"> S[<a id="jvalue5">j</a>]= <a id="sjvalue2"></a></span>
</p>
```

Figure 34. HTML of the swap animation step

There are many <div> elements written and meticulously indexed with distinct id or class values, but have no visual effect on the webpage. These invisible elements are used by the Stage 3 equivalent of pauserunanim() – stage3animation(). Although the swapping appears to be one step in the animation sequence in the web demo, on the code level the stage3animation() have actually run through seven steps: from step 9 to step 15 (which means from the step in which stage3stepcount = 8, to the step in which stage3stepcount = 14). Figure 35 shows the code of these seven steps:



```
case 8: stage3scrolltosi(); break;
case 9: if ( !stage3step8started ){
            step8anim = stage3step8slot1();
            elementsi.style.opacity = "0";
            swapslot1.style.opacity = "1";
            step8anim.play();
            stage3step8started = true;
        }
        else { step8anim.play(); }
        break;
case 10: stage3scrolltosj(); break;
case 11:  stage3step8started = false;
        if ( !stage3step9started ){
            step9anim = stage3step9slot2();
            elementsj.style.opacity = "0";
            swapslot2.style.opacity = "1";
            step9anim.play();
            stage3step9started = true;
        }
        else { step9anim.play(); }
        break;
case 12: stage3step9started = false;
        if ( !stage3step10started ){
            step10anim = stage3step10slot1();
            step10anim.play();
            stage3step10started = true;
        }
        else { step10anim.play(); }
        break;
case 13:  stage3scrolltosi(); break;
case 14:  stage3step10started = false;
        if ( !stage3step11started ){
            step11anim = stage3step11slot2();
            step11anim.play();
            stage3step11started = true;
        }
        else { step11anim.play(); }
        break;
```

Figure 35. stage3animation() code section of the swap animation

Some lines of the code also appear as “parent code section” when this section details the functions and their corresponding steps described in 3.3.2 Abstract Flow of Swap Animation.

Swap animation implementation is the most challenging part of the web demo of RC4. A great length of code is written just to animate the swapping of S[i] and S[j] in Stage 3. In Stage 2, the swap animation is written with the same basic animation methods but uses fewer lines of code. only one function – stage2swapstep1() is needed to realize the swap animation. The tradeoff is that the sub-steps in the swapping are not pausable. Does it make the user experience obviously less interactive?

Chapter 4

Conclusions and Future Work

4.1 Conclusions

This report documents the implementation of RC4 Web Demonstration which is to be used as teaching materials in introductory cryptography course in NTU. The web demo is to be freely available to anyone who takes an interest in cryptography and pseudo-random number generators.

As the web demo is written in thousands of lines of code, the report leaves out the elementary knowledge and straightforward functions used in the overall implementation, but focuses on analyzing the building of complex, multi-phased animations. It walks the reader through the implementations of two important animations in the web demo: the sequential web animation and the swap animation. It also touches on how to integrate the two animation techniques to achieve more advanced animation effects.

Notable pitfalls and failures occurred during the implementation are discussed so that they may be avoided in the future.

4.2 Recommendation in Future Work

The web demo models RC4 algorithm completely but not perfectly. More efforts can be put into visualization and beautification of the web interface. There are a few areas of improvement which have high priority. One is the Stage 2 swap animation which is implemented differently from the Stage 3 one. Further work could be done to reimplement it to offer a more interactive user experience. Another area of improvement would be code reduction and simplification. Many JavaScript functions have similar effects. Stage 2 and Stage 3 of the web demo use two distinct sets of functions to realize almost the same animations. last but not the least, providing the user with random-key generation in Stage 1 could give users an easier, smoother experience.

Throughout the project, APIs or examples of sophisticated animations are not encountered. Perhaps animation techniques developed and discussed in this report can be further generalized and developed into a web-animation library for future works.

References

[1] "Dictionary.com Definitions Cryptography," [Online]. Available: https://www.dictionary.com/browse/cryptography. [Accessed 17 November 2018].

[2] M. Rouse, "Definition cryptography," [Online]. Available: https://searchsecurity.techtarget.com/definition/cryptography. [Accessed 16 November 2018].

[3] N. G. McDonald, "Past, Present, And Future Methods Of Cryptography And Data Encryption".

[4] W. Stallings, Cryptography and Network Security Principles and Practices, 4th ed., vol. I, Prentice Hall, 2005, p. 30.

[5] "Public Key Cryptography," International Business Machines Corporation, [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSB23S_1.1.0.13/gtps7/s7pkey.html. [Accessed 16 November 2018].

[6] "Steganography," Merriam-Webster, Incorporated, [Online]. Available: https://www.merriam-webster.com/dictionary/steganography. [Accessed 16 November 2018].

[7] W. Stallings, Cryptography and Network Security Principles and Practices, 4th ed., vol. I, Prentice Hall, 2005, p. 64.

[8] "Symmetric Cryptography," International Business Machines Corporation, [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSB23S_1.1.0.13/gtps7/s7symm.html. [Accessed 18 November 2018].

[9] W. Stallings, Cryptography and Network Security Principles and Practices, 4th ed., vol. I, Prentice Hall, 2005, pp. 23-194.

[10 A. Datta, "CryptoIntroduction," Singapore, 2018.

[11 A. Datta, "Random Cryptography Demos," Nanyang Technological University, 2018. [Online]. Available: http://pdcc.ntu.edu.sg/sands/cryptodemos/#fh5co-tab-feature-vertical7. [Accessed 18 November 2018].

[12 jgrahamc, "RC4 Algorithm".

[13 V. Gagrani, "Presentation over Cryptographic Primitives (RC4) (Personal-Portfolio)," [Online]. Available: https://www.youtube.com/watch?v=KM-xZYZXElk. [Accessed 22 November 2018].

[14 S. Sriadhi, R. Rahim and A. S. Ahmar, "RC4 Algorithm Visualization for Cryptography Education," in *2nd International Conference on Statistics, Mathematics, Teaching, and Research*, 2018.

[15 "RC4 – Symmetric Ciphers Online," [Online]. Available: http://rc4.online-domain-tools.com/. [Accessed 22 November 2018].

[16 "HTML DOM classList Property," w3schools.com, [Online]. Available: https://www.w3schools.com/jsref/prop_element_classlist.asp. [Accessed 22 November 2018].

[17 "Animationend Event," w3schools.com, [Online]. Available: https://www.w3schools.com/jsref/event_animationend.asp. [Accessed 22 November 2018].

[18 "The CSS Animation Events," Kirupa, 14 April 2013. [Online]. Available: https://www.kirupa.com/html5/css_animation_events.htm. [Accessed 22 November 2018].

[19 f. e. al., "Element.animate()," 31 August 2018. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Element/animate. [Accessed 22 November 2018].

[20 ".animate()," The jQuery Foundation, [Online]. Available: http://api.jquery.com/animate/. [Accessed 18 November 2018].

[21 "Cipher," Cambridge University Press, [Online]. Available: https://dictionary.cambridge.org/dictionary/english/cipher. [Accessed 16 November 2018].

[22 M. Asaad, "RC4 Security Algorithm".