INTERNATIONAL BACCALAUREATE EXTENDED ESSAY

COMPUTER SCIENCE

A Comparison Of Shuffling Algorithms Used In Music Players

Fisher-Yates/Knuth Shuffle vs Naive Shuffle

Research Question: To what extent does the Fisher-Yates/Knuth algorithm outperform the Naive algorithm when shuffling a music playlist?

Word Count: 3767

Table of Contents

1.0 Introduction

Shuffling is an action that we may experience on a day to day basis, whether it happens with a deck of cards or within our favourite playlist. The action of shuffling is taking a sorted array of elements and rearranging them randomly. Take, for example, an array of songs that are stored in a music player, when the "shuffle play" mode is activated, it will consider the ordered songs as inputs, generating an output that will be any random permutation of the input, such that all permutations are equally likely to occur. Music makes up for a big part of my day, I listen to songs on my playlist continuously and sometimes I enjoy activating the shuffle play mode to remind me of older songs that I have saved on my playlist.

A couple of months ago, I stumbled across an interesting article that claimed that Spotify's shuffle wasn't actually random. This article, in particular, struck me because I had been questioning Spotify's shuffling algorithm for a while as I noticed that primarily songs from my favourite artists were the ones being played even while using the shuffle play mode. By reading a few articles and studies I learned that the reason for this was that Spotify, like other music players, uses "predictive music shuffling" , basically shuffling songs based on the user's preferences. I was now determined and interested in finding out more about shuffling algorithms and which one is more "random". One of the first and best known shuffling algorithms is the

"Fisher-Yates shuffle". There have been many variations of this particular shuffle, as it has been adapted to computer programming, some of these are the: Durstenfeld/Knuth Shuffle, the Sattolo's shuffle and the Naive shuffle.

The aim of this Extended Essay is to investigate and compare two well known shuffling algorithms: The Fisher-Yates-Knuth Shuffle and the Naive Shuffle. These will be implemented, run and tested to generate results which will be analysed in order to understand which shuffling algorithms is most effective. In order to do this, the java code will be written in a code editor called "Sublime Text", while the spreadsheet from Google Sheets will be used to record and graph results. Lastly, I will be shuffling 50 songs from my own Spotify playlist in order to test the randomness of Spotify's shuffling algorithm compared to the shuffling algorithms discussed throughout this essay.

1.1 Research Question

The research question that will be tackled is: "To what extent does the Fisher-Yates/Knuth algorithm outperform the Naive algorithm when shuffling a music playlist?"
This research question is worthy of investigation not only because it deepens one's understanding of shuffling algorithms but also because it allows the exploration of the validity and reliability of this topic.

2.0 Investigation

2.1 Fisher-Yates/Knuth Shuffle

2.1.1 Theory

The Fisher Yates Shuffle, also known as the Knuth Shuffle, is named after Ronald Fisher and Frank Yate (1938), and was later on modernised by Richard Durstenfled and Donald Knuth. It is an algorithm that is used for generating a random and an unbiased permutation of a finite sequence. Biased sampling occurs when a sample is collected in a certain way that results in some members of an intended set of items to have a lower or a higher sampling probability than other items, resulting in biased (non-random) sample. The strength of the Fisher Yates shuffle is that the random permutation generated from the shuffle has the same probability of occurring as another, and still valid, permutation. In mathematics, permutations are the various ways in which objects from a set may be selected, sometimes to form subsets. A basic example of this would be:

Consider a set of 4 different objects such as: A,B,C,D

There are 24 (given by 4 factorial "!") different permutations of these four items if they are all considered as the same time:

ABCD, **ABDC**, **ACBD**, **ACDB**, **ADBC**, **ADCB**,

BACD, **BADC**, **BCAD**, **BCDA**, **BDAC**, **BDCA**,

CABD, **CADB**, **CBAD**, **CBDA**, **CDAB**, **CDBA**,

DABC, **DACB**, **DBAC**, **DBCA**, **DCAB**, **DCBA**

Meanwhile, there are 12 different permutations ($_4P_2$) if these four objects were taken 2 at a time:

AB, BA, AC, CA,

AD, DA, BC, CB,

BD, DB, CD, DC

Thus the formula to find out the amount of different permutations that can occur for a given set of objects is:

$_nP_k = n!/(n - k)!$, where n is the number of objects in the set and k is the number of objects taken at a time.

The exclamation point (!) stands for "factorial" which indicated that all consecutive positive integers from 1 up to "n" (included) are multiplied together ("Permutations And Combinations | Description, Examples, & Formula").

The original implementation described by Fisher and Yates was featured in their publication of "Statistical tables for biological, agricultural and medical research". The method for generating a random permutation was the following:

The first step was to write down all the numbers from 1 through N, (being the amount of objects in the set)

Range	**Random Number**	**Set of numbers**	**Result**
1-8		1, 2, 3, 4, 5, 6, 7, 8	

Secondly, pick a random number k, between the range of numbers, for example k=4, and write it down as the first object of the permutation.

Range	**Random Number**	**Set of numbers**	**Result**
1-8	4	1, 2, 3, ~~4~~, 5, 6, 7, 8	4

This step has to be repeated until all numbers are striked out and a random permutation is generated.

Range	**Random Number**	**Set of numbers**	**Result**
1-7	6	1, 2, 3, ~~4,~~ 5, 6, ~~7,~~ 8	4, 7,
1- 6	2	1, ~~2~~, 3, ~~4~~, 5, 6, ~~7~~, 8	4, 7, 2
1-5	3	1, ~~2,~~ 3, ~~4,~~ ~~5~~, 6, ~~7~~, 8	4, 7, 2, 5
1-4	4	1, ~~2~~, 3, ~~4,~~ ~~5,~~ 6, ~~7, 8~~	4, 7, 2, 5, 8
1-3	3	1, ~~2,~~ 3, ~~4,~~ ~~5,~~ ~~6,~~ ~~7,~~ ~~8~~	4, 7, 2, 5, 8, 6
1-2	1	~~1,~~ ~~2,~~ 3, ~~4,~~ ~~5,~~ ~~6,~~ ~~7,~~ ~~8~~	4, 7, 2, 5, 8, 6, 1
1	1	~~1, 2, 3, 4, 5, 6, 7, 8~~	4, 7, 2, 5, 8, 6, 1, 3
Final Answer			**4, 7, 2, 5, 8, 6, 1, 3**

This method will generate an unbiased permutation from any set of items.

2.1.2 The Modern Implementation of the Fisher-Yates Algorithm

In 1964 Richard Durstenfeld introduced a modernized version of the Fisher-Yates algorithm, which was later popularized by Donald Knuth in a monograph called "The Art of Computer

Programming”. The changes between this method of shuffling and the original one mainly reduced the time complexity of the algorithm by not needing to count and recognize the items in the set that were not yet struck out *(Medium)*. It was implemented as an in-place shuffle, meaning that the elements in the array are shuffled in place rather than generating another rearranged version of the array. This was done by swapping the last item of the set with the item that had been struck out. We will be considering the same set of numbers used in the previous method:

Range	**Random Number**	**Set of Numbers**	**Result**
1- 8		1, 2, 3, 4, 5, 6, 7, 8	

Another variation from the “Original Method” is that the struck out item will always be placed first in the result.

Range	**Random Number**	**Set of Numbers**	**Result**
1- 8	4	1, 2, 3, 8, 5, 6, 7	4
1- 7	6	1, 2, 3, 8, 5, 7	6, 4
1- 6	2	1, 7, 3, 8, 5	2, 6, 4
1- 5	3	1, 7, 5, 8	3, 2, 6, 4
1- 4	4	1, 7, 5	8, 3, 2, 6, 4
1- 3	3	1, 7	5, 8, 3, 2, 6, 4
1- 2	1	7	1, 5, 8, 3, 2, 6, 4
Final Answer			**7, 1, 5, 8, 3, 2, 6, 4**

Thus, by using the same set of items and the same pattern of random numbers the two permutations will look differently:

Final Permutation from the Original Method: {**4, 7, 2, 5, 8, 6, 1, 3**}

Final Permutation from the Modern Method: {**7, 1, 5, 8, 3, 2, 6, 4**}

2.1.3 Pseudocode

```
To shuffle an array a[n-1] of n elements (indices 0 … n-1)

for i from 0 to n-1 do
j = random integer with 0 <= j <= i

Exchange a[j] with a[i]

End loop
```

2.1.4 JAVA

```
import java.util.*;

public class Knuth {

public static void knuth(int[] array){

      Random r = new Random();
      for(int i = array.length - 1; i > 0; i--){
          int index = r.nextInt(i+1);

      //swap
           int tmp = array[index];
           array[index] = array[i];
           array[i] = tmp;
      }
}

      public static void main(String[] args) {

           int [] array = {1,2,3,4,5,6,7,8,9,10,11,12};
           System.out.println("Before: " + Arrays.toString(array));

           knuth(array);
```

```
            System.out.println("After: " + Arrays.toString(array));
        }
}
```

As the mathematician Robert R. Coveyou once stated: “The generation of random numbers is too important to be left to chance” *("Fisher-Yates Shuffle")*. The java algorithm above imports the “java.util” package, which provides the Random() class. This class is used to generate pseudo-random numbers in java. In most cases, the algorithm of the random number generator works with a seed value, however, if it is not provided, the seed number is created by the system *(Geeksforgeeks)*. An advantage of this class is that it provides numerous methods that generate random numbers of different types, such as: integer, long, double etc [**APPENDIX A**] *(Geeksforgeeks)*. The method “nextInt(int n)” returns a pseudorandom int value between 0 and the specified value, from the random number generator. In this case n = i + 1, thus array.length + 1 ultimates in n = 12. The random number is represented by the r variable of type Random which will be used in the index of the array to swap a certain element in order to start shuffling the array *("Java Random - Journaldev")*.

In order to swap items in the array a third variable of type integer is used. This is a temporary variable named “tmp” that stores the item that needs to be swapped. The item that is stored is found at the array at the index r.nextInt(i+1), meaning that the index of the item is randomly generated by the Random( ) method. Consider the following example using the JAVA code above:

If r.nextInt(i+1) = 4, number “5” will be the item that needs to be swapped with another item, thus stored temporarily inside the variable “tmp”. This is denoted by:

int tmp = array[index]

The second step is to actually swap item number “5” with another item. This item is contained in array [i] which is the last item in the array. This occurs in the following way:

array[index] = array[i]

array[i] = tmp

The last item in the array will be stored in array[index] where item number “5” was once stored, while “5” (which is being temporarily stored inside the tmp variable) will be stored as the last item of the array, at array[i]. The for loop that is used at the beginning of the code serves to repeat the action of swapping for as many times as the number of items contained in the array. For every repetition, however, the value of I will continue to decrease (i--), meaning that the item number “5” which has already been swapped, cannot be swapped anymore, and every other item swapped will go in front of the items that have already been swapped. Overall, given a collection of items that needs to be randomly sorted, the random number generator will select one of the “unshuffled” items and swap it with the last item in the collection that has not yet been selected *(Medium)*. This will continue until there are no remaining “unshuffled” items” . In this case there would be 12! different permutations all equally likely to occur.

Example of the output in Terminal

```
javac Knuth.java
java Knuth

Before: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
After: [3, 11, 10, 2, 6, 4, 5, 1, 7, 12, 8, 9]

java Knuth

Before: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
After: [10, 6, 11, 9, 4, 5, 3, 12, 8, 7, 2, 1]
```

A major advantage of the Fisher-Yates/Knuth Shuffle Algorithm is that it has an optimal linear time efficiency (Roberts). The execution of an algorithm is usually measured with the time and the memory needed given the size of a collection. The computer implementation of the original shuffling method of Fisher-Yates, has an algorithm time complexity of $O(n^2)$, as it takes the extra time and memory while trying to calculate the remaining items that are still "unshuffled". On the other hand, the original method outlined within the Knuth Algorithm has a time complexity of $O(n)$ *("Fisher-Yates Shuffle")*. Lastly, another one of the primary benefits of the Knuth Shuffle is that it is an "in-place algorithm", meaning that the swapping will occur within an existing collection rather than in two seperate collections (Jones).

2.2 The Naive Shuffle Algorithm

2.2.1 Theory

Another shuffling algorithm that has been used several times in the past, for example with online poker games, is the Naive Shuffle *(Roberts)*. A Naive algorithm in general is a very straightforward solution to a problem, usually simple to devise and implement but can often consume large amounts of resources such as time and memory *(Atwood)*. Even though it might be difficult to detect, the code for the naive shuffle is actually broken, since when it is implemented and run, it will lead to a biased result. As mentioned previously, the shuffling of items will result in one output that will be any random permutation of the input, such that all permutations are equally likely to occur. However, what seems to happen when the Naive shuffling algorithm is implemented, run and tested is that certain random permutations are more likely to occur than others, making the entire shuffle biased and unfair. The explanation for this biased result only requires some basic math. For a fair shuffle of any array of length n there are n! possible permutations, however in the Naive shuffle there are n^n possible code paths. The reasoning behind this shuffle is that each of the "n" elements in the array can be swapped with any of the other "n" possible elements, including itself. Let's take for example a deck of 4 cards for which the total number of permutations would be 4! or 24. If the Naive algorithm is used to shuffle these four cards it will end up generating 4^4 or 256 non-unique outcomes or code paths. It is clearly impossible for the 24 possible shuffles to be represented by 256 code paths, meaning the naive shuffle contains a bias, which will expand as the number of items in the array being shuffled also increases *(Atwood)*.

2.2.2 Pseudocode

```
for (int i = 0; i < items.Length; i++) {
int n = rand.Next(items.Length);
  Swap(ref items[i], ref items[n]);
}
End loop
```

The pseudocode above represents the Naive shuffling algorithm, which simply loops through the array, swapping each item with another randomly chosen item in the array.

2.2.3 JAVA

```
public class Naive {

public static void Naive(int[] array){

      Random r = new Random();
      for(int i = array.length - 1; i > 0; i--){
          int index = r.nextInt(i);

        //swap
      int tmp = array[index];
      array[index] = array[i];
      array[i] = tmp;
 }

}
```

The problem with the Naive shuffle, is that instead of outputting a shuffled combination that is equally likely to occur as any other possible combination, certain combinations are over-represented, making the whole process biased *(Medium)*. This occurs because, as stated previously, the algorithm would output a total of n^n possible non-unique outcomes, instead of n! unique combinations. Another main disadvantage of the Naive Shuffle, compared to the Knuth Shuffle is its time complexity, which is O(n^2) *(Atwood)*. This is because Naive algorithms tend to consume larger amounts of resources such as time, space and memory.

The Naive shuffle has been implemented in real applications such as certain online poker games. There are other issues and aspects within software security which won't be covered in this essay, however if the following example of a shuffling algorithm used in online poker is taken into consideration a few bugs can be noticed:

```
for i:=1 to 52 do begin
r := random(51) + 1;
swap := card[r];
card[r] := card[i];
card[i] := swap;
end;
```

The first problem with this code is that the random( ) method never gets to, in this case, 52 which means that there is no possible combination in which the last card will end up in the 52nd place. Random number generators, used in shuffling algorithms, can in fact lead to common implementation errors. Moreover the shuffle is not uniform as it is randomly generating and picking any item from the set instead of focusing only on the items that are still unshuffled. According to Professor Sedgewick, from the University of Princeton: “The random() uses a 32-bit seed” thus there can only be 2^{32} possible shuffles which is less compared to the actual possible shuffles that should occur with a 52 card deck (5 factorial). *(Roberts)*

3.0 Testing

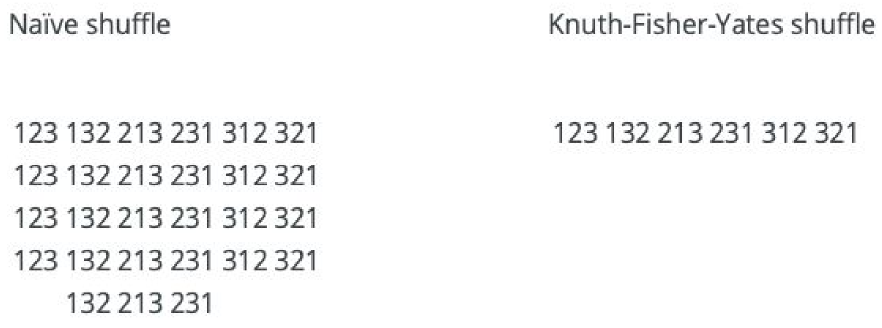
To test both the Knuth Shuffling Algorithm and the Naive Shuffle, as well as record results, for "n" amount of times, the following code was used:

Algorithm for testing

```
public static void main(String[] args) {

        //possible combinations are:
        //1,2,3 = a
        //1,3,2 = b
        //2,1,3 = c
        //2,3,1 = d
        //3,1,2 = e
        //3,2,1 = f

        int a, b, c, d, e, f = 0;
        int sumA = 0;
        int sumB = 0;
        int sumC = 0;
        int sumD = 0;
        int sumE = 0;
        int sumF = 0;

        int k = 0;

        int [] array = {1,2,3};

        while(k < 1000) {

        knuth(array);

        if(array[0]==1 && array[1]==2){
                int newsumA = sumA + 1;
                sumA = newsumA;
        }

        if(array[0]==1 && array[1]==3){
                int newsumB = sumB + 1;
```

```
            sumB = newsumB;
        }

        if(array[0]==2 && array[1]==1){
            int newsumC = sumC + 1;
            sumC = newsumC;
        }

        if(array[0]==2 && array[1]==3){
            int newsumD = sumD + 1;
            sumD = newsumD;
        }

        if(array[0]==3 && array[1]==1){
            int newsumE = sumE + 1;
            sumE = newsumE;
        }

        if(array[0]==3 && array[1]==2){
            int newsumF = sumF + 1;
            sumF = newsumF;
        }
             k = k+1;
        }
            System.out.println("{1,2,3} = " + sumA);
            System.out.println("{1,3,2} = " + sumB);
            System.out.println("{2,1,3} = " + sumC);
            System.out.println("{2,3,1} = " + sumD);
            System.out.println("{3,1,2} = " + sumE);
            System.out.println("{3,2,1} = " + sumF);
        }
```

In this case, the array containing numbers "1, 2 and 3" is being shuffled 1000 times. All possible permutations of a 3-item array generated by the Knuth Shuffle and the Naive Shuffle can be represented in the following way:



Naïve shuffle	Knuth-Fisher-Yates shuffle
123 132 213 231 312 321	123 132 213 231 312 321
123 132 213 231 312 321	
123 132 213 231 312 321	
123 132 213 231 312 321	
132 213 231	

Therefore, the Naive Shuffle will have 27 non-unique outcomes, while the Knuth Shuffle will have 6 unique outcomes, resulting in certain combinations to be over-represented when the items were shuffled with the Naive algorithm *("Dan's Random Thoughts")*.

4.0 Results and Discussion

The results obtained by testing the algorithms described previously will be represented and displayed in the form of tables and bar graphs.

Table 1: Knuth Shuffle after 1000 times

Permutations	Shuffle
abc	154
acb	164
bac	183
bca	151
cab	175
cba	173
Total	1000

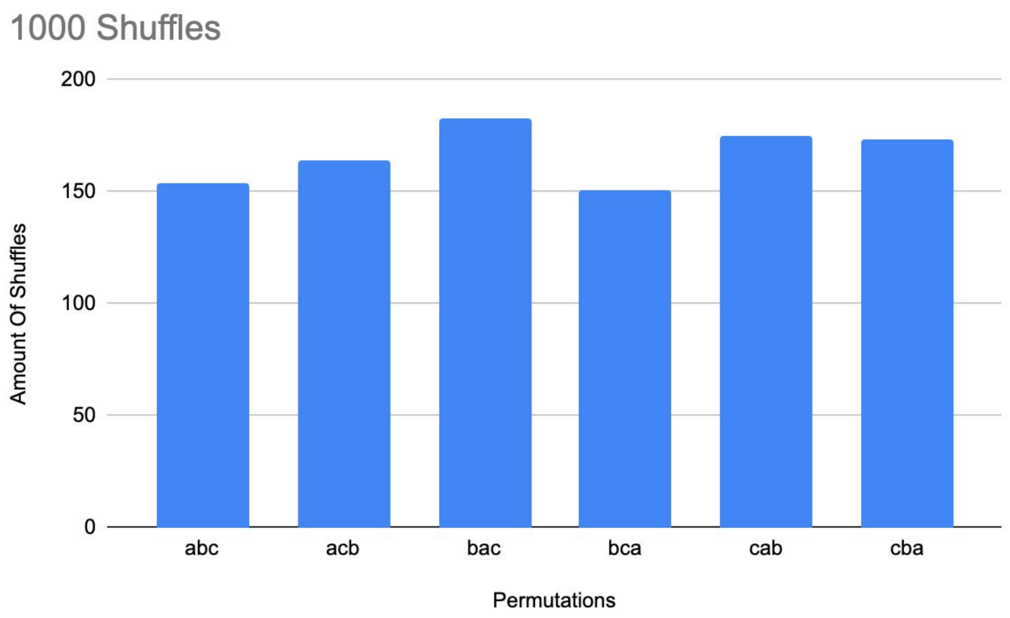


Table 2: Knuth Shuffle after 10000 times

Permutations	Shuffle
abc	1694
acb	1648
bac	1705
bca	1682
cab	1656
cba	1615
Total	10000

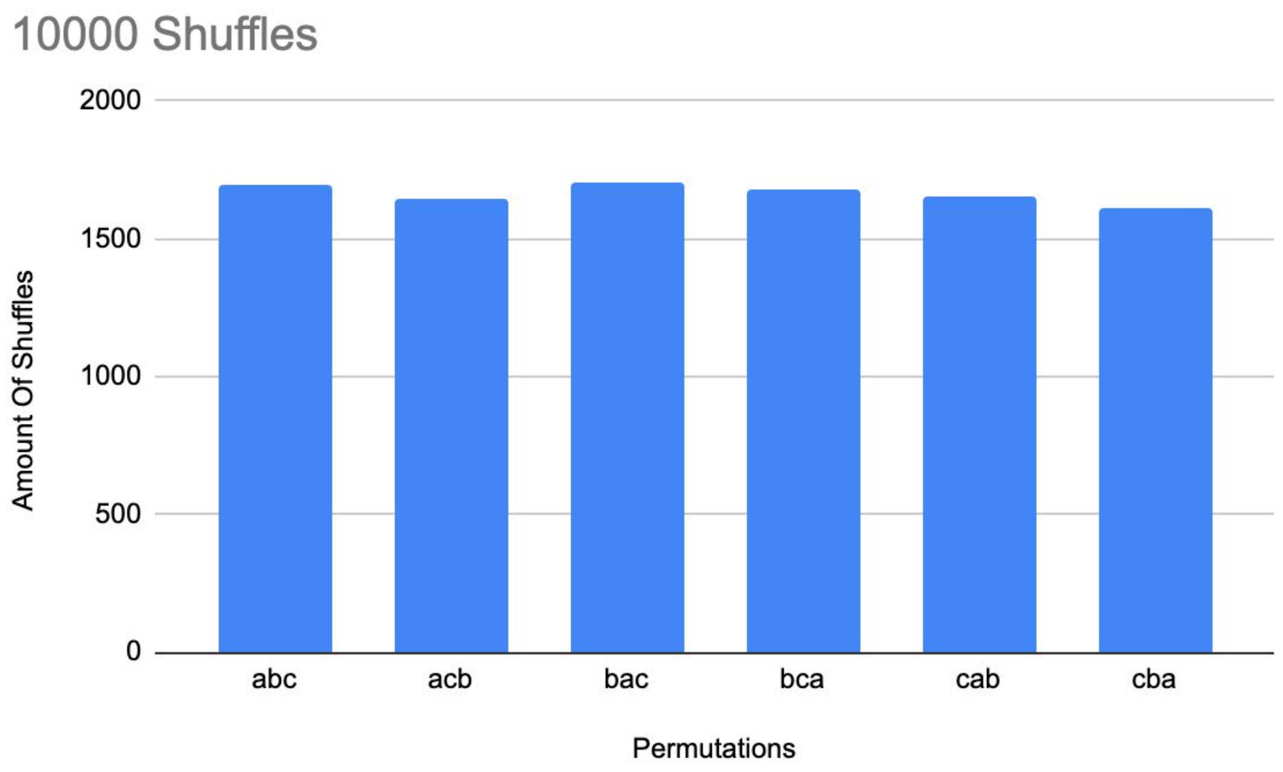
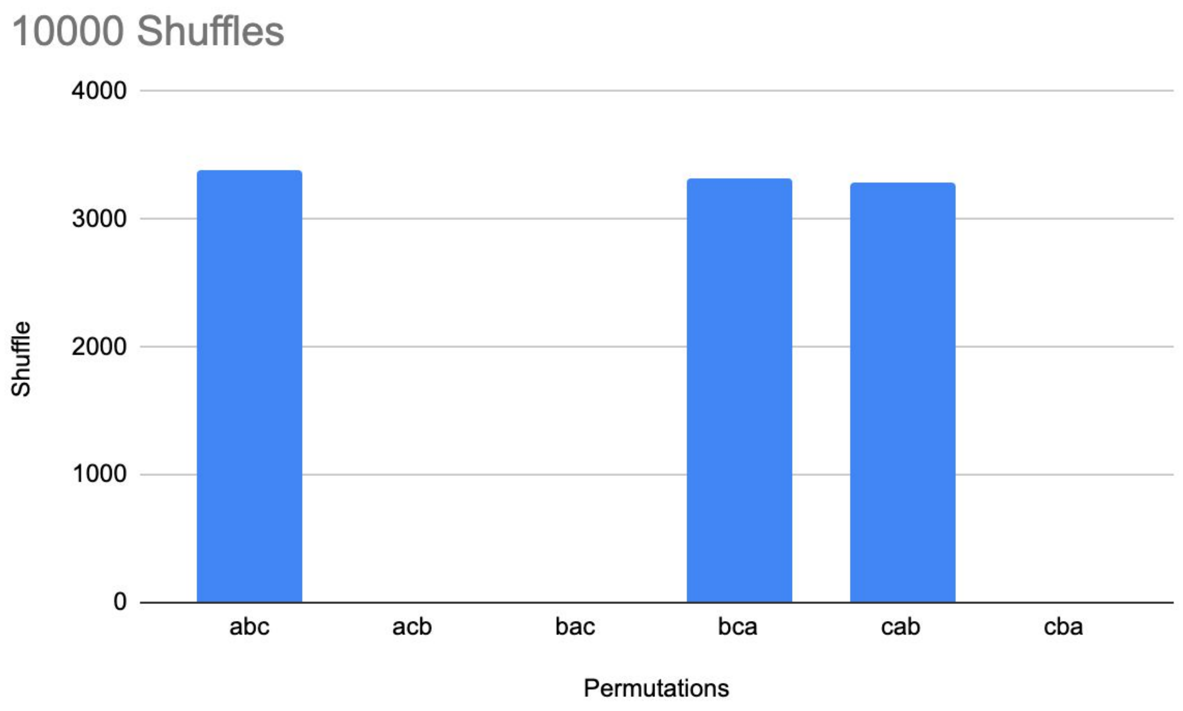


Table 3: Naive Shuffle after 10000 times

Permutations	Shuffle
abc	3384
acb	0
bac	0
bca	3321
cab	3295
cba	0
Total	10000



Tables 1 and 2 display the amount of times that a certain permutation of a set of items a, b and c has been outputted after running the Knuth algorithm 1000 and 10000 times respectively. It can be observed that, to a certain extent, the amount of times that a certain permutation was outputted, is similar to the total amount of times that a different permutation of the same set of elements has been outputted. Moreover, the data also shows that as the amount of times that the algorithm is run increases, the more uniformly distributed are the outcomes.

On the other hand, Table 3 displays the outcomes of the Naive Shuffle Algorithm. As it can be observed from the data, permutations abc, bca and cab are overwhelmingly over-represented, while permutations acb, bac and cba have never been outputted. As described in the previous paragraphs, for a set containing 3 elements, there are only 3! = 6 unique combinations, which is significantly smaller than the $3^3 = 27$ "paths" considered by the Naive Shuffle *("Dan's Random Thoughts")*. With an even bigger set of elements, for example shuffling a 6 card deck, the difference between the outcomes from the two algorithms would be much greater *(Medium)*. Therefore, with 27 paths and only 6 outputs, it is clear that some combinations are dangerously over represented. This shows that the Naive shuffling algorithm used throughout this essay and described in the previous paragraphs is "completely" broken as, after 1000 shuffles, only 3 out of the 6 possible permutations were represented. It is also important to note that there exists more than one "Naive" shuffling algorithm, which all contain certain implementation errores, some easier and some harder to detect. Another example of a Navie Shuffling Algorithm is described in an article called "*Comparing a Naive Shuffle Algorithm with the Fisher-Yates Shuffle*" from *Observable Notebook (Observablehq).* (**APPENDIX B**)

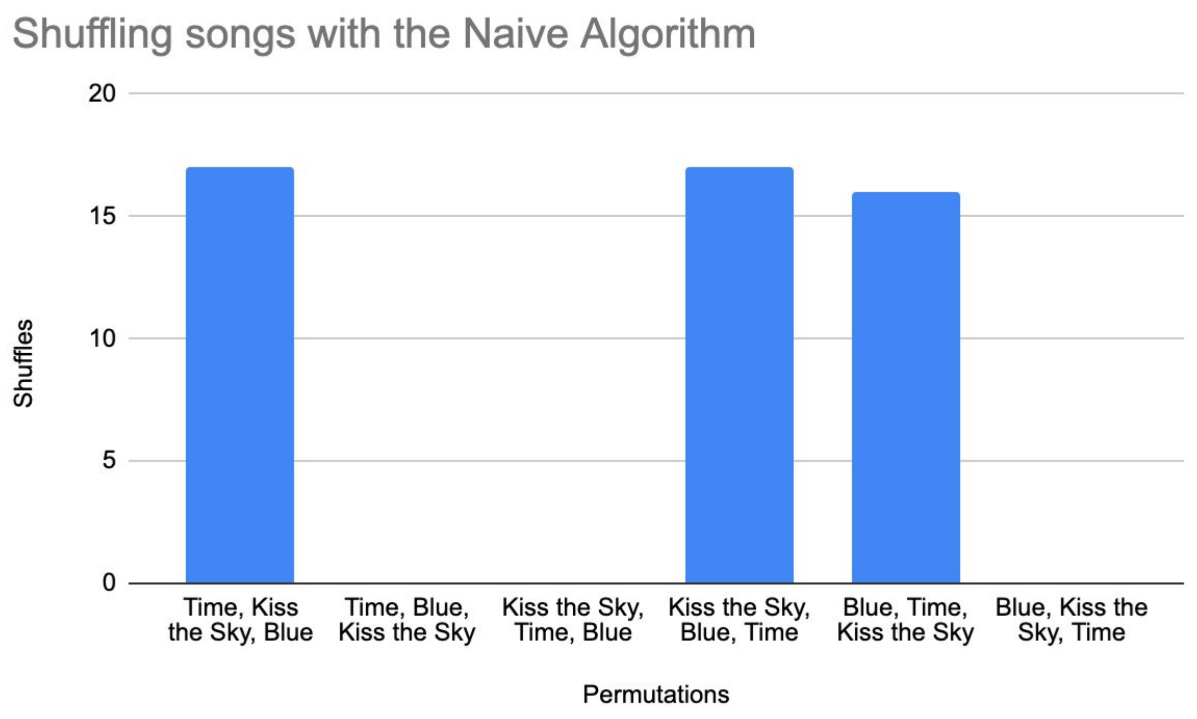
5.0 Shuffling Songs

In order to further connect with the research question tackled, this essay attempts to shuffle three songs using the Naive, the Knuth Shuffle Algorithm, and *Spotify's* random Shuffle.

5.1 Shuffling Songs using the Naive Algorithm

Table 4: Songs shuffled using the Naive Algorithm

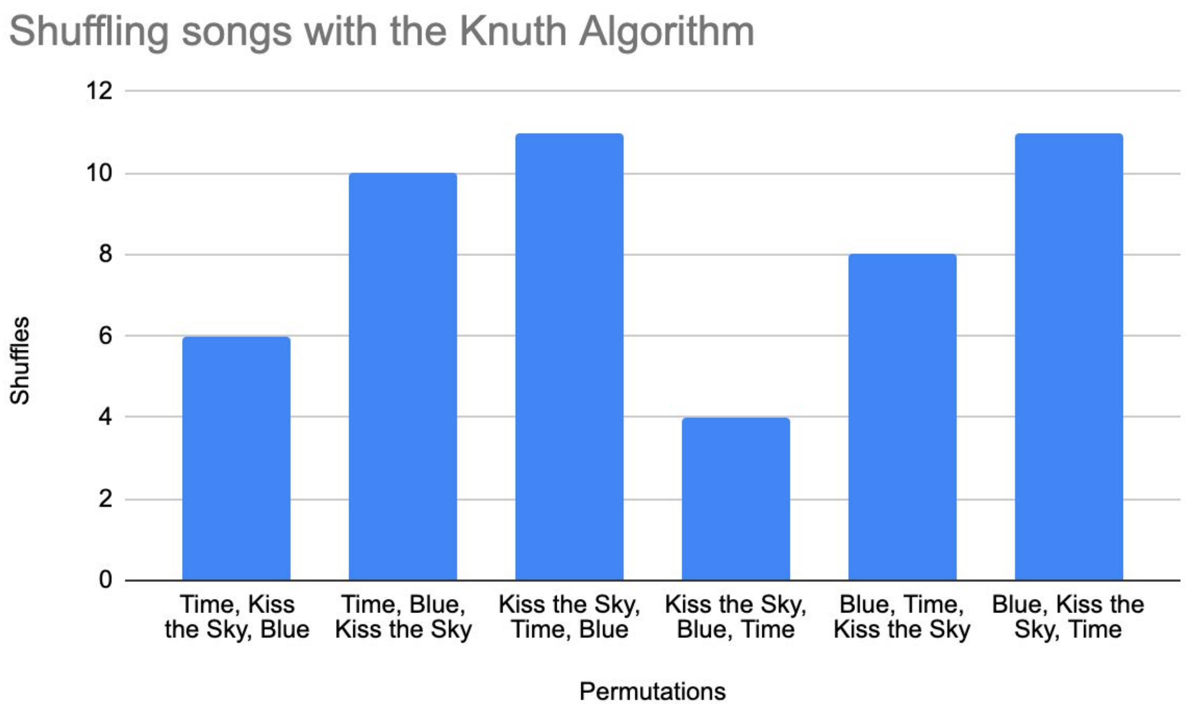
Permutations	**Shuffle**
Time, Kiss the Sky, Blue	**17**
Time, Blue, Kiss the Sky	**0**
Kiss the Sky, Time, Blue	**0**
Kiss the Sky, Blue, Time	**17**
Blue, Time, Kiss the Sky	**16**
Blue, Kiss the Sky, Time	**0**
Total	50



5.2 Shuffling Songs using the Knuth Algorithm

Table 5: Songs shuffled using the Knuth Algorithm

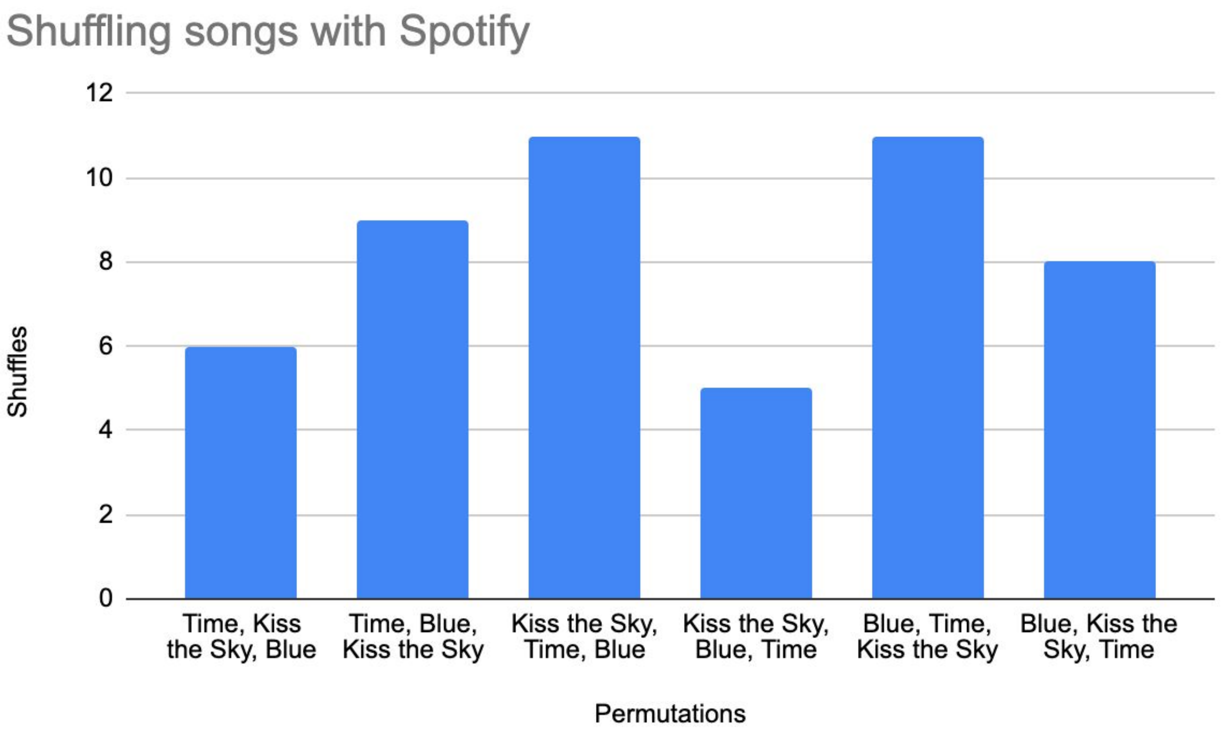
Permutations	**Shuffle**
Time, Kiss the Sky, Blue	**6**
Time, Blue, Kiss the Sky	**10**
Kiss the Sky, Time, Blue	**11**
Kiss the Sky, Blue, Time	**4**
Blue, Time, Kiss the Sky	**8**
Blue, Kiss the Sky, Time	**11**
Total	50



5.2 Shuffling Songs with Spotify

Table 6: Songs shuffled using Spotify

Permutations	**Shuffle**
Time, Kiss the Sky, Blue	**6**
Time, Blue, Kiss the Sky	**9**
Kiss the Sky, Time, Blue	**11**
Kiss the Sky, Blue, Time	**5**
Blue, Time, Kiss the Sky	**11**
Blue, Kiss the Sky, Time	**8**
Total	50



Tables 4, 5 and 6 show the total amount of times a certain permutation was outputted, after shuffling 3 songs (Time, Kiss the Sky and Blue), for 50 times. Table 4 shows outcomes generated by the Naive Shuffle. In the same way as it occured in Table 3, only half of the possible permutations were represented by this shuffle. On the other hand, table 5 shows the outcomes generated by the Knuth Shuffle. Even with this shuffle, it can be noticed that certain permutations are more represented than others, for example the pattern "*Blue, Kiss the Sky, Time*" is significantly more represented than "*Kiss the Sky, Blue, Time*". However, if Table 5 and Table 2 are compared, it is also fair to deduce that, as the amount of shuffles increases, the amount of permutations represented will be more uniformly distributed. Lastly, Table 6 represents the outcomes generated by *Spotify's* Random Shuffle. Similarly to the Knuth Shuffle, certain permutations are more represented than others, thus after 50 shuffles, all outcomes are not uniformly distributed. On the other hand however, it can be quickly noticed that Spotify's random shuffle is significantly similar to the Knuth Shuffle, while completely different from the Naive Shuffle. Interestingly, most outcomes from both Table 5 and 6 are similar, and in some cases they are exactly the same, for example, in both tables, two permutations were represented 11 times, and as it can be noticed in their respective charts, for both shuffling algorithms, the permutation "Kiss the Sky, Blue, Time" was the least represented.

6.0 Conclusion

The investigation carried out in the Essay had the aim of addressing and answering the following research question: “To what extent does the Fisher-Yates/Knuth algorithm outperform the Naive algorithm when shuffling a music playlist?”.

The Knuth Shuffle Algorithm uses an elegant, well structured and documented shuffling technique. Most importantly it is an in-place shuffle, thus it is more time efficient, with a time complexity of O(n), and takes up low amounts of space and memory *("Fisher-Yates Shuffle")*.

On the other hand, the Naive Shuffle Algorithm proposes a straight-forward technique to shuffle a given set of elements, however, even though it can be difficult to detect where, the algorithm is fundamentally broken. The outputs are biased and non-unique, as certain combinations are over-represented.

Therefore, the results obtained provide evidence to support the claim that the Fisher-Yates/Knuth shuffling algorithm outperforms the Naive shuffling algorithm, as it guarantees the output of a unique and unbiased combination, of a given array of elements, that is equally likely to occur as any other possible combination. Moreover, it can also be concluded that, to a certain extent, the Random Shuffle used in the music player, *Spotify*, is significantly close to the Knuth Algorithm. This is possible to change if a larger set of items had been used, as Spotify’s random shuffle is said to adopt a “predictive music shuffling”. In fact, for a larger set of items, which was not represented throughout the investigation, that Naive shuffle could possibly represent more than 3 possible permutations of the set, and the difference between the Naive and the Knuth Shuffle would be visually more evident *(Medium)*.

6.0.1 Evaluation

This essay certainly acknowledges that there are limitations of the methodology used throughout the investigation. Firstly the Naive Shuffle algorithm that was used is one of many, which in this case was significantly biased. Moreover, as stated in the previous paragraphs, all shuffling algorithms are greatly impacted by the size of the set that will be shuffled. Because of this, the essay also attempts to find patterns between the Knuth Shuffle and Spotify's Shuffle when shuffling a set of 30 songs. The results, however, are inconclusive as more knowledge and information, which was not distributed to the public by *Spotify*, regarding their algorithm is needed (**APPENDIX C** and **APPENDIX D**). One of the most important aspects that had to be considered was the random number generator used within the algorithms, as it can easily lead to implementation errors and bugs. In fact, this essay lacks knowledge of which Random number generator is used by *Spotify*, making the final comparison between Spotify's Shuffle and the Knuth Shuffle inconclusive. Lastly, other than the Random Number generator used by Spotify, this music player is part of a billion dollar company and a very complex and important industry, thus it can be fair to deduce that its algorithm is more complex than the Knuth Algorithm and is not completely represented within this Essay.

Appendices

APPENDIX A: Random Number Generator

```
import java.util.Random;

public class RandomNumberExample {

     public static void main(String[] args) {

          //initialize random number generator
          Random random = new Random();

          //generates boolean value
          System.out.println(random.nextBoolean());

          //generates double value
          System.out.println(random.nextDouble());

          //generates float value
          System.out.println(random.nextFloat());

          //generates int value
          System.out.println(random.nextInt());

          //generates int value within specific limit
          System.out.println(random.nextInt(20));

     }
}
```

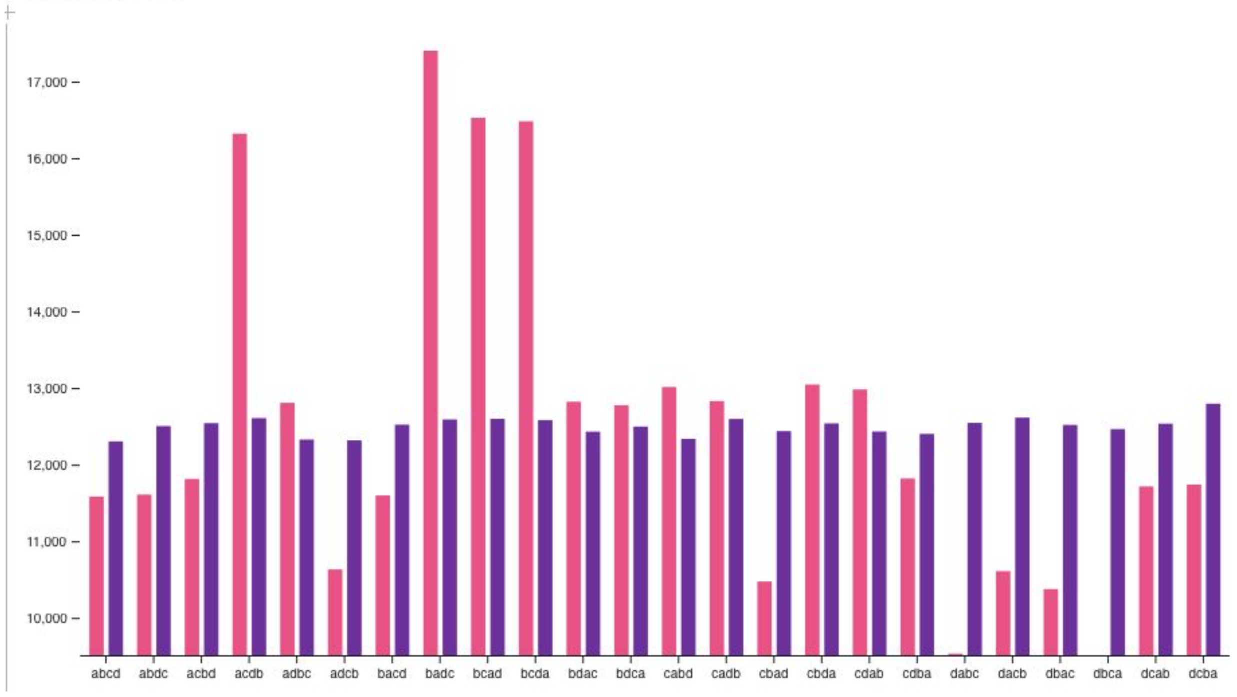
Terminal (Example):

```
true
0.404842815877433
```

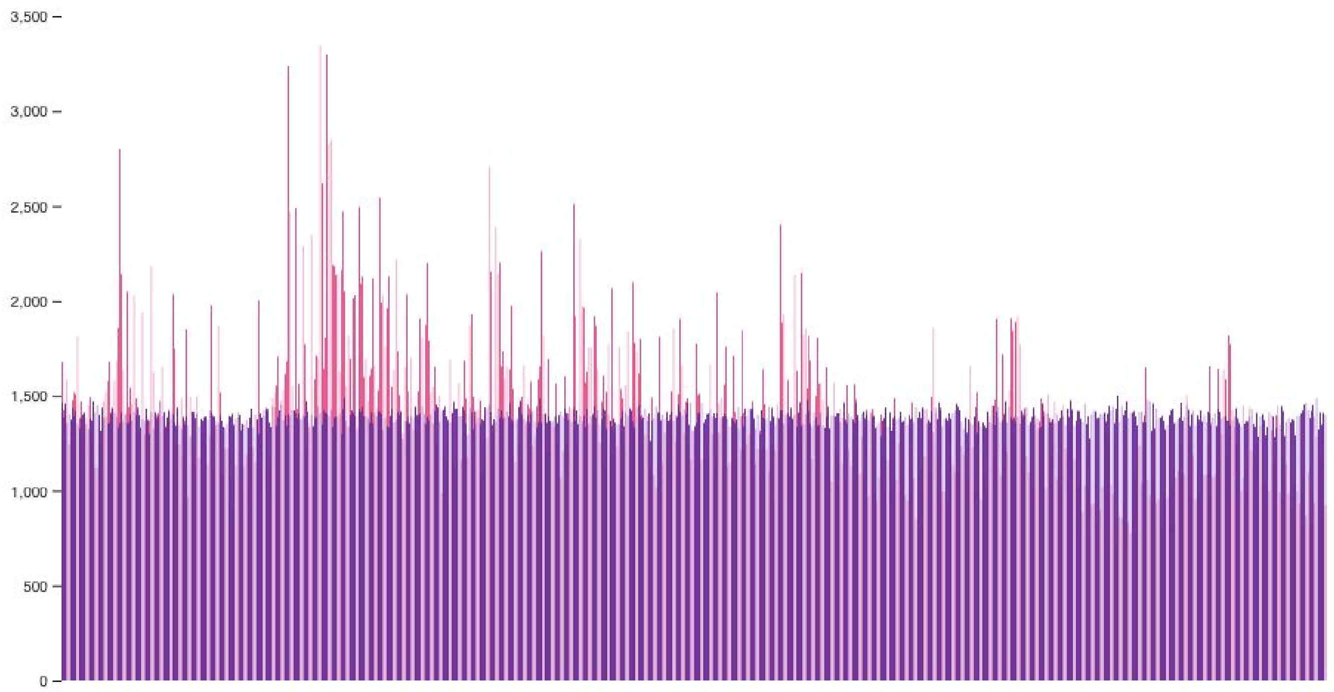
```
0.99694717
1979557935
19
```

APPENDIX B : Comparing a Naive and the Fisher-Yates Shuffle

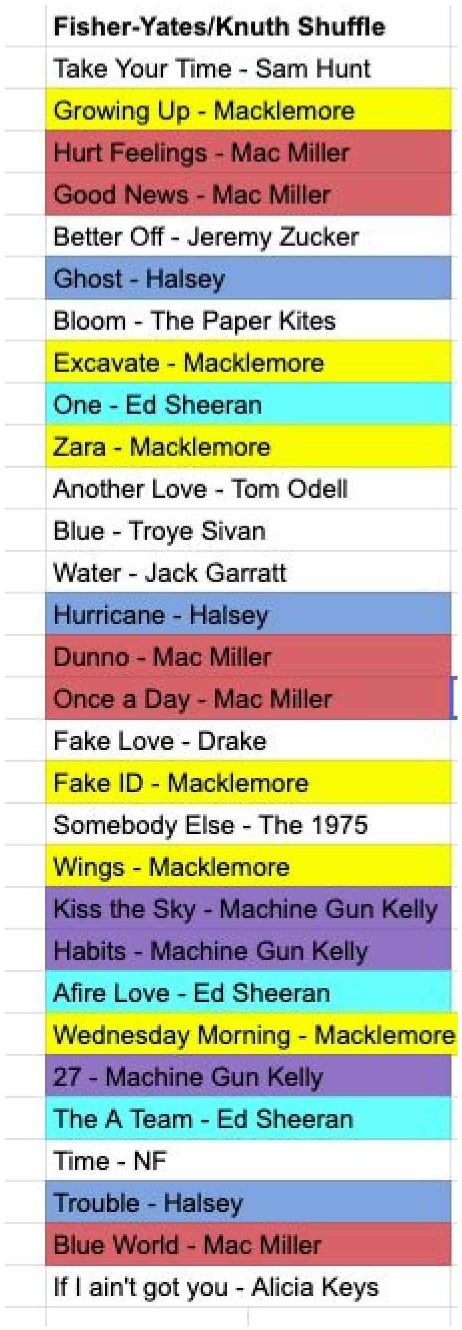
Naive shuffle vs. Fisher-Yates shuffle, 300k tries on a 4-item set



Naive shuffle vs. Fisher-Yates shuffle, 1m tries on a 6-item set

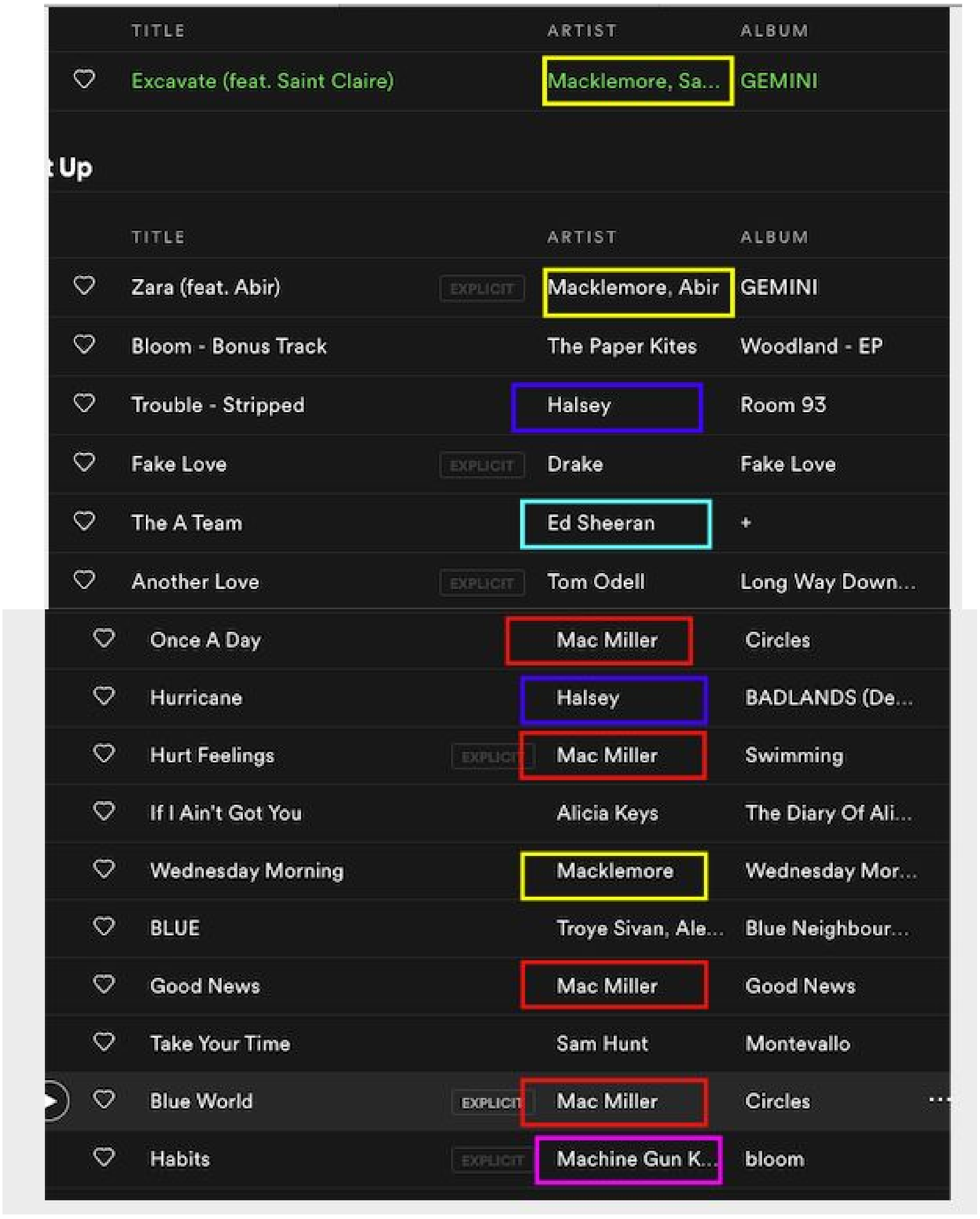


APPENDIX C: Shuffling 30 Songs using the Knuth Algorithm



Fisher-Yates/Knuth Shuffle
Take Your Time - Sam Hunt
Growing Up - Macklemore
Hurt Feelings - Mac Miller
Good News - Mac Miller
Better Off - Jeremy Zucker
Ghost - Halsey
Bloom - The Paper Kites
Excavate - Macklemore
One - Ed Sheeran
Zara - Macklemore
Another Love - Tom Odell
Blue - Troye Sivan
Water - Jack Garratt
Hurricane - Halsey
Dunno - Mac Miller
Once a Day - Mac Miller
Fake Love - Drake
Fake ID - Macklemore
Somebody Else - The 1975
Wings - Macklemore
Kiss the Sky - Machine Gun Kelly
Habits - Machine Gun Kelly
Afire Love - Ed Sheeran
Wednesday Morning - Macklemore
27 - Machine Gun Kelly
The A Team - Ed Sheeran
Time - NF
Trouble - Halsey
Blue World - Mac Miller
If I ain't got you - Alicia Keys

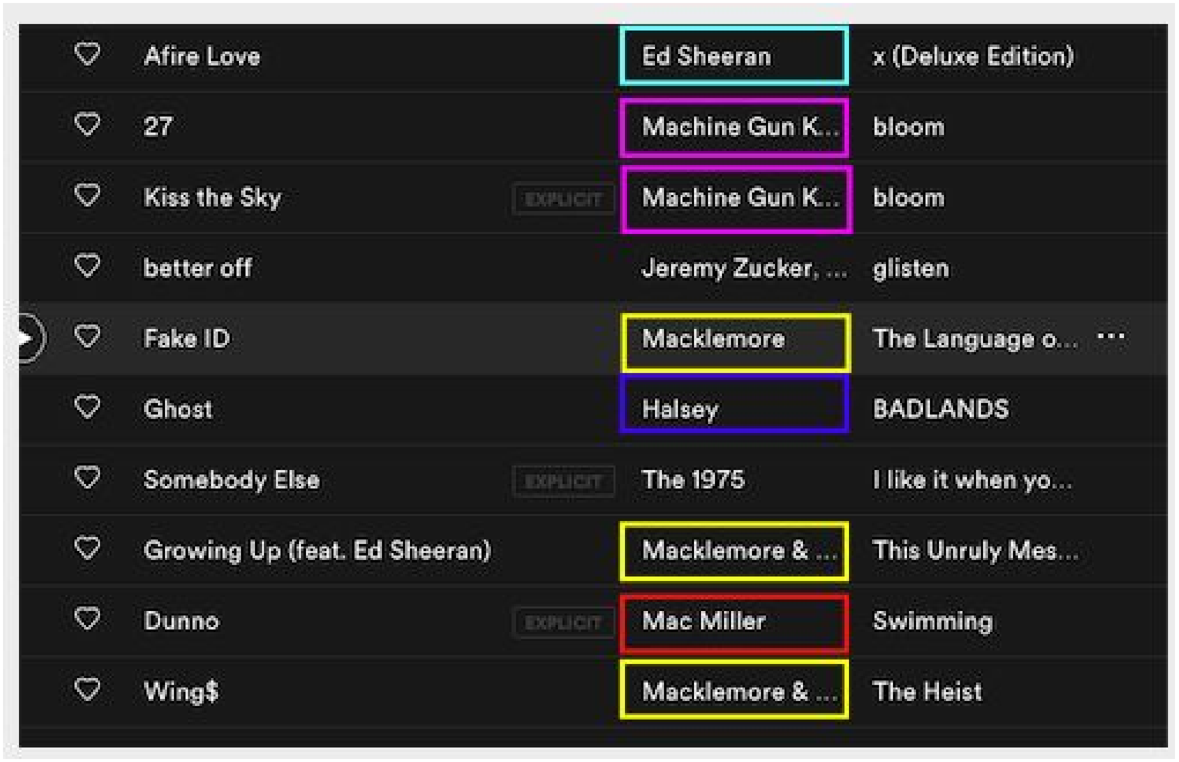
APPENDIX D: Shuffling 30 songs using *Spotify's* Random shuffle



TITLE		ARTIST	ALBUM
Excavate (feat. Saint Claire)		Macklemore, Sa...	GEMINI

Up

TITLE		ARTIST	ALBUM
Zara (feat. Abir)	EXPLICIT	Macklemore, Abir	GEMINI
Bloom - Bonus Track		The Paper Kites	Woodland - EP
Trouble - Stripped		Halsey	Room 93
Fake Love	EXPLICIT	Drake	Fake Love
The A Team		Ed Sheeran	+
Another Love	EXPLICIT	Tom Odell	Long Way Down...
Once A Day		Mac Miller	Circles
Hurricane		Halsey	BADLANDS (De...
Hurt Feelings	EXPLICIT	Mac Miller	Swimming
If I Ain't Got You		Alicia Keys	The Diary Of Ali...
Wednesday Morning		Macklemore	Wednesday Mor...
BLUE		Troye Sivan, Ale...	Blue Neighbour...
Good News		Mac Miller	Good News
Take Your Time		Sam Hunt	Montevallo
Blue World	EXPLICIT	Mac Miller	Circles
Habits	EXPLICIT	Machine Gun K...	bloom



Afire Love		Ed Sheeran	x (Deluxe Edition)
27		Machine Gun K...	bloom
Kiss the Sky	EXPLICIT	Machine Gun K...	bloom
better off		Jeremy Zucker, ...	glisten
Fake ID		Macklemore	The Language o...
Ghost		Halsey	BADLANDS
Somebody Else	EXPLICIT	The 1975	I like it when yo...
Growing Up (feat. Ed Sheeran)		Macklemore & ...	This Unruly Mes...
Dunno	EXPLICIT	Mac Miller	Swimming
Wing$		Macklemore & ...	The Heist

Bibliography

Algorithm, Complexity et al. "Complexity Of The Fisher-Yates Shuffle Algorithm". *Theoretical Computer Science Stack Exchange*, 2020, https://cstheory.stackexchange.com/questions/26/complexity-of-the-fisher-yates-shuffle-algorithm.

Atwood, Jeff. "The Danger Of Naïveté". *Coding Horror*, 2020, https://blog.codinghorror.com/the-danger-of-naivete/.

"Comparing A Naive Shuffle Algorithm With The Fisher-Yates Shuffle". *Observablehq.Com*, 2020, https://observablehq.com/@oldwestaction/comparing-a-naive-shuffle-algorithm-with-the-fisher-yates-s.

"Dan's Random Thoughts". *Dyladan.Me*, 2020, https://dyladan.me/abc/2016/01/20/shuffle/.

"Fisher-Yates Shuffle". *Xlinux.Nist.Gov*, 2020, https://xlinux.nist.gov/dads/HTML/fisherYatesShuffle.html.

"Java Random - Journaldev". *Journaldev*, 2020, https://www.journaldev.com/17111/java-random.

"Java.Util.Random Class In Java - Geeksforgeeks". *Geeksforgeeks*, 2020, https://www.geeksforgeeks.org/java-util-random-class-java/.

Jones, Matthew. "Understanding The Fisher-Yates Shuffling Algorithm". *Exception Not Found*,2016, https://exceptionnotfound.net/understanding-the-fisher-yates-card-shuffling-algorithm/.

"Permutations And Combinations | Description, Examples, & Formula". *Encyclopedia Britannica*, 2020, https://www.britannica.com/science/permutation.

"Randomness Is Hard: Learning About The Fisher-Yates Shuffle Algorithm & Random Number Generation". *Medium*, 2020, https://medium.com/@oldwestaction/randomness-is-hard-e085decbcbb2.

Roberts, Martin. "The Fisher-Yates Algorithm | Extreme Learning". *Extremelearning.Com.Au*, 2020, http://extremelearning.com.au/fisher-yates-algorithm/.