De Bruijn and Balanced Cut-down Sequences

Luke Fischer *

April 23, 2022

Abstract

This document highlights an overview of de Bruijn sequences and their variants. One variant in particular is the balanced cut-down de Bruijn sequence. This de Bruijn-like sequence allows for constructions of arbitrary length cyclic strings while also holding desirable randomness properties.

1 Introduction

The University of Guelph had installed a cipher lock system for McLaughlin Library to improve overnight security. The door will open when presented with the correct four-digit code. This means there are 10,000 four-digit combinations and to try each distinct four-digit combination would require a burglar to make 40,000 key strokes. However, the lock system has a major security flaw. The door will open whenever the correct four-digit code is the last four digits of any sequence as described in [1]. Now the burglar only has to make one key stroke to attempt a new four-digit code instead of four. If the burglar is smart, he or she will be certain to open the door after 10,003 key strokes and on average 5,000 key strokes.

Consider if the library key pad only consisted of numbers 0 to 2 and required a 3-digit passcode. This means there are 27 different possible 3-tuple codes for a burglar to try. If the burglar used the sequence (wrapping back to the front after reaching the end):

0012200222102121101112020 10

he or she would be guaranteed to open the door after 29 key strokes and an average of 13.5 key strokes. Do sequences like this always exist? Can we construct a sequence for the orginal four-digit code comprised of an alphabet $\mathbf{A}$ where $\mathbf{A} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$? Yes, this is *a de Bruijn sequence*. Luckily, Professor Sawada had begun studying these sequences and advised the library to install a more sophisticated lock system.

Given an alphabet $\mathbf{A}$ of size K, a de Bruijn sequence of order N on $\mathbf{A}$ is a cyclic string in which every possible length-N string on $\mathbf{A}$ appears exactly once as a substring. The sequence above in bold is an example of a de Bruijn sequence of order $N = 3$, on an alphabet $\mathbf{A} = \{0, 1, 2\}$. Observe that the sequence contains every possible string of length-N on $\mathbf{A}$ exactly once as a substring:

001 012 122 220 200 002 022 222 221 210 102 021 212 121 211 110 101 011 111 112 120 202 020 201 010 100 000.

*CIS4900, University of Guelph, Canada

Well known properties of de Bruijn sequences[2]

Given a de Bruijn sequence $\mathcal{D}$ of order N on an alphabet $\mathbf{A}$ of size K, we can say $\mathcal{D}$:

1. Is of the length K^N.
2. Contains the same number instances of each character in $\mathbf{A}$ (balanced).
3. Contains an equal number of continuous runs of each character in $\mathbf{A}$ of the same length (run property).
4. Contains every distinct length-N string on $\mathbf{A}$ as a substring exactly once (span-N property).

Another interesting property of de Bruijn sequences is that given a sequence of order N on an alphabet $\mathbf{A}$ of size K both its length and the number of possible substrings of length N on $\mathbf{A}$ is K^N. This is an advantageous property because if you wanted to store every length N string on an arbitrary alphabet you would save more than $\approx N$ times space by constructing a de Bruijn sequence rather than an array of strings.

Example 1 Consider the following de Bruijn sequence $\mathcal{D}$ of order $N = 5$ on the alphabet $\mathbf{A} = \{0, 1\}$. Let's verify that it satisfies all the properties of a de Bruijn sequence

$$\mathcal{D} = 00001111100010100110111010110010.$$

1. $|\mathcal{D}| = K^N = 32$.

2. Number of 1's = 16, number of 0's = 16.

3. Length 1 runs: 1's = 16, 0's = 16
   Length 2 runs: 1's = 8, 0's = 8.
   Length 3 runs: 1's = 4, 0's = 4.
   Length 4 runs: 1's = 2, 0's = 2.
   Length 5 runs: 1's = 1, 0's = 1.

4. Contains each distinct length 5 binary string as a substring exactly once: {00000, 00001, ..., 11110, 11111}.

All four properties are satisfied; therefore, this is in fact a binary de Bruijn sequence.

Now that we have a basic understanding of de Bruijn sequences we can discuss how to construct them in Section 2. Also, what if we want to construct a sequence of arbitrary length while still holding desirable properties of de Bruijn sequences? Variants such as the balanced cut-down de Bruijn sequence will be discussed in Section 3. An implementation of the generation and verification of binary balanced cut-down de Bruijn sequences can be found in the Appendix A and B respectively.

2 Constructing de Bruijn Sequences

Now that we have a foundation of the properties of de Bruijn sequences it is important to know how to constuct them. The website: http://www.debruijnsequence.org/ [3] describes varieties of ways to construct de Bruijn sequences and universal cycles. Below are instructions and examples regarding the construction of the prefer-largest and recursive techniques.

2.1 Prefer-Largest Construction

The prefer-largest construction belongs to the greedy class of constructions. This class of constructions include some of the simpler methods to understand but come at the cost of requiring exponential space. The prefer-largest construction generates the lexicographically largest de Bruijn sequence. As follows:

1. Initialize a linear string with the seed 0^{N-1}.
2. Append the largest symbol from **A** that does not create a duplicate length-N substring. Repeat until no new symbol can be added.
3. Remove the seed.

Example 2 A prefer-largest construction for a simple de Bruijn sequence D, where $A = \{0, 1\}$, and $N = 3$.

1. Initialize: 00
2. Append: 001 → 0011 → 00111 → 001110 → 0011101 → 00111010 → 001110100 → 0011101000
3. Remove seed: ~~00~~11101000 → **11101000**

Although we used a linear to string to construct the sequence, the end result is a de Bruijn sequence.

The prefer-smallest construction is identical to the prefer-largest construction but instead has the smallest symbol replaced by K - 1, the second smallest replaced by K - 2 and so on. For example, if we constructed the de Bruijn sequence above using the prefer-smallest construction the final result would be 00010111.

2.2 Recursive Construction for Binary de Bruijn Sequences

Given a de Bruijn sequence $\mathcal{S}$ of order N we can recursively generate a de Bruijn sequence $\mathcal{T}$ of order $N + 1$ by applying Lempel's lift and join[4].

2.2.1 Lempel's Lift and Join on a Binary Alphabet

Lempel's lift and join is a recursive approach to transform a de Bruijn sequence $\mathcal{S}$ of order N to a de Bruijn sequence $\mathcal{T}$ of order $N + 1$. This is done by constructing two strings λ_0 and λ_1 from $\mathcal{S}$. Given a de Bruijn sequence $\mathcal{S}$ of order $N > 0$, we can construct a de Bruijn sequence $\mathcal{T}$ of order $N + 1$ by following these steps:

1. Begin with the first character in $\mathcal{S}$.
2. Add all characters up to and including the current character in $\mathcal{S}$.
3. Add the subscript of the lambda you are constructing (0 or 1) to the result of step two and apply modulo 2 onto this result.
4. Append the result from step 3 to the respected string (λ_0 or λ_1).
5. Increment the character index and go to step two unless $\mathcal{S}$ has been completely iterated through.
6. If $\mathcal{S}$ has an odd weight (has an odd number of 1's) append $\overline{\lambda_i}$ to λ_i.
   6.1 Assign $\mathcal{T}$ to λ_0.
7. Else, assign $\mathcal{T}$ to the result of joining strings λ_0 and λ_1. See "Joining Arbitrary Cyclic Strings" in Section 2.2.2 for instructions.

2.2.2 Joining Arbitrary Cyclic Strings

Joining cyclic strings together is the process of generating a de Bruijn sequence $\mathcal{T}$ of order $N+1$ from $\mathcal{S}$ of order N by joining λ_0 and λ_1 at a common length-N prefix. To construct a de Bruijn sequence $\mathcal{D}$ from λ_0 and λ_1:

1. Identify the common length-N prefix.
2. Begin at the first character of prefix λ_0 and iterate through until a full loop of the cyclic string is made. Each time copying the current symbol from λ_0 to $\mathcal{D}$.
3. Apply step 2 but now iterate and copy λ_1 to $\mathcal{D}$.

2.2.3 Putting It Together

Now that we have an understanding of Lempel's lift, let's recursively generate a de Bruijn sequence of order $N = 5$.

Example 3 Given a de Bruijn sequence $\mathcal{D}1 = 10$ of order 1 we will recursively construct a de Bruijn sequence $\mathcal{D}5$ of order 5 using Lempel's lift and join.

$N = 1 \rightarrow N = 2$: $\lambda_0 = 11$, $\lambda_1 = 00$. $\mathcal{D}2 = 1100$.

$N = 2 \rightarrow N = 3$: $\lambda_0 = \mathbf{1}000$, $\lambda_1 = \mathbf{0}111$. $\mathcal{D}3 = 10001011$.

$N = 3 \rightarrow N = 4$: $\lambda_0 = 1111\mathbf{001}0$, $\lambda_1 = 00\mathbf{001}101$. $\mathcal{D}4 = 0010111100110100$.

$N = 4 \rightarrow N = 5$: $\lambda_0 = 0011010111011\mathbf{000}$, $\lambda_1 = 110010\mathbf{000}100111$.

$$\mathcal{D}5 = 10000011010111011000100111110010.$$

Note: Only a de Bruijn sequence of order $N = 1$ will have to use step 6 in section 2.2.1. However, Lempel's lift is also used to construct arbitrary length de Bruijn-like sequences such as the Balanced Cut-down sequence outlined in Section 3. Step 6 will prove to be more impactful when working with these types of sequences.

3 Variations of de Bruijn Sequences

Variations of de Bruijn sequences are similar to de Bruijn sequences but with some key differences. They are similar because variations also try to construct cyclic strings generating some set of distinct strings of a length N on an arbitrary alphabet $\mathbf{A}$, but are different because they lack the de Bruijn sequence property of $L = K^N$. Some variations include the *cut-down de Bruijn sequence* and the *generalized de Bruijn sequence*.

A cut-down de Bruijn sequence is an arbitrary length-L cyclic string where $1 \leq L \leq K^N$ such that, every length-N string on $\mathbf{A}$ appears at most once [5]. On the other hand, a generalized de Bruijn sequence is a cyclic string such that every length-$\lfloor log_K L \rfloor$ string on $\mathbf{A}$ is a substring of the sequence at least once and every length-$\lceil log_K L \rceil$ string on $\mathbf{A}$ appears in the sequence at most once [6]. *A balanced cut-down de Bruijn sequence* is an arbitrary length sequence such as the cut-down de Bruijn sequence and shares similar substring repetition properties as the generalized de Bruijn sequence. However, the balanced cut-down de Bruijn sequence goes a step further to ensure no two length-M strings where $1 \leq M \leq L$ on $\mathbf{A}$ differentiate in more than 1 repetition. This reduces bias of the inclusion or exclusion of one length-M string in the sequence from another. This improves the sequence's randomness property and therefore, makes it more de Bruijn-like than the generalized and cut-down de Bruijn sequence. See Section 3.2 where the balanced cut-down de Bruijn sequence is discussed in more depth.

Note: the "balanced cut-down de Bruijn sequence" is being used to refer to the $P_L^{(K)}$ sequence introduced by Nellore and Ward [7].

3.1 Differentiating de Bruijn-like Sequences

In this section we use examples to better understand the difference between the sequences discussed in Section 3.

Example 4 A Cut-down That Is Not a Generalized

Consider the following length-23 cut-down sequence $\mathcal{D}$ of order $N = 2$ on $\mathbf{A} = \{0, 1, 2, 3, 4, 5, 6, 7\}$

$$06050403020141312332211.$$

This is a cut-down sequence of length-23 because every length-N string on $\mathbf{A}$ occurs at most once in the sequence. However, it is clearly not a generalized de Bruijn sequence because the length-$\lfloor log_K L \rfloor$ string "7" on $\mathbf{A}$ is not included in the sequence.

Example 5 A Generalized That Is Not a Balanced Cut-down

Consider the following length-23 generalized de Bruijn sequence $\mathcal{D}$ on $\mathbf{A} = \{0, 1, 2\}$

$$00220210200120112111010.$$

This is a generalized de Bruijn sequence because each length-$\lfloor log_K L \rfloor$ string on $\mathbf{A}$ appears in the sequence at least once and each length-$\lceil log_K L \rceil$ string on $\mathbf{A}$ appears at most once.

However, this is not a balanced cut-down de Bruijn sequence because the repetitions of the character "0" and "2" differentiate by 3.

3.2 The Balanced Cut-down de Bruijn Sequence

A balanced cut-down de Bruijn sequence is a length-L cyclic string on an alphabet $\mathbf{A}$ of size K such that the number of occurences of any length-M string on $\mathbf{A}$ for $M \leq L$ as a substring of the sequence is $\lfloor L/K^M \rfloor$ or $\lceil L/K^M \rceil$. It is trivial to see that when $L = K^N$ for any postive integer N then $\lfloor L/K^M \rfloor = \lceil L/K^M \rceil$ is simply a de Bruijn sequence. This is why we are more interested in constructing balanced cut-down sequences where $L \neq K^N$. Now that we have discussed the desirable properties of the balanced cut-down de Bruijn sequence let's see how to construct one.

3.2.1 Constructing a Binary Balanced Cut-down de Bruijn Sequence

To construct a binary balanced cut-down de Bruijn sequence we will follow a procedure similar to the recursive construction in Section 2.2. We will use Lempel's lift and join from Sections 2.2.1 and 2.2.2 but with modifications to the joining technique. Now, instead of joining λ_0 and λ_1 at an arbitrary common length-N prefix we will use joining techniques described in the GENERATEP2L algorithm [7]. Finally, at the end of each lift and join iteration we will extend the sequence by the character "1" depending on the desired length of the balanced cut-down sequence. Here is the construction of a length 37 balanced cut-down sequence on a binary alphabet following the GENERATEP2L algorithm [7].

Example 6

Preliminaries
For $L = 37$ we derive $N = 6$ from $N = \lceil log_2 L \rceil$.

Compute the digits (d_i) of the base 2 representation of L. $d = 100101$.

Initialize cyclic string $\alpha = 1$.

The Process
for $j = 1$ to N - 1 **do**
    if α has an odd weight **then**
        $\alpha = \lambda_0$
    else
        Follow the correct joining techniques outline in GENERATEP2L[7]
    extend by d_j characters at 1^{j-1}

$j = 1$: **lift** $\alpha = 1$ to $\lambda_0 = 10$, $\lambda_1 = 01$. Weight of α is odd. $\alpha = \lambda_0$.

$j = 2$: **lift** $\alpha = 10$ to $\lambda_0 = 1100$, $\lambda_1 = 0011$. Weight of α is odd. $\alpha = \lambda_0$.

$j = 3$: **lift** $\alpha = 1100$: $\lambda_0 = 1000$ and $\lambda_1 = 0111$. 1^{j-1} is a substring of α so we **join** λ_0 and λ_1 at 0^{++}_{j-1} giving us $\alpha = 01000111$. Extend by d_j characters at 1^{j-1}. $\alpha = 010001111$.

$j = 4$: **lift** $\alpha = 010001111$: $\lambda_0 = 011110101100001010$ and $\lambda_1 = 100001010011110101$. Weight of α is odd. $\alpha = \lambda_0$.

$j = 5$: **lift** $\alpha = 011110101100001010$: $\lambda_0 = 0101001101111100111010110010000011 00$ and $\lambda_1 = 1010110010000011000101001101111100 11$. Weight of α is odd. $\alpha = \lambda_0$. Extend by d_j characters at 1^{j-1}.

$$\alpha = \mathbf{0101001101111110011101011001000001100}$$

Note: the notation 0^{++}_{j-1} corresponds to the string (0 mod K, (0 + 1) mod K, (0 + 2) mod K, ..., (0 + j - 1) mod K).

An implementation of a binary balanced cut-down sequence constructor in the C programming language can be found in Appendix A. To verify a sequence of binary characters holds the properties of a balanced cut-down sequence see the C program in Appendix B. See section 4 where we discuss the details of writing these programs.

4 Binary Balanced Cut-down Implementation Details

Constructing and verifying binary balanced cut-down de Bruijn sequences was not a straight forward task. Sections 4.1 and 4.2 discusses the obstacles faced while writing these programs and how they were overcome.

4.1 Constructing a Binary Balanced Cut-down Sequence

The implementation for a binary balanced cut-down constructor stemmed off the pseudocode description from the algorithm GENERATEP2L[7]. However, this algorithm had two major flaws. Firstly, it would only generate balanced cut-down sequences of length 2^Z where Z is an integer ≥ 0. And, as discussed earlier, a balanced cut-down that shares this length property is a de Bruijn sequence, rendering GENERATEP2L[7] obsolete. To fix this problem a technique was used which Nellore and Ward [7] discussed elsewhere in their paper where at the end of each iteration the character "1" is periodically inserted into the sequence. The details of this insertion are described in Section 3.2.1.

Secondly, when generating a binary balanced cut-down de Bruijn sequence of length 2^Z where Z is an integer ≥ 0 GENERATEP2L[7] would generate a sequence of length 2^{Z-1}. To fix this the for loop conditional statement was modified to $j < N$ - 1 + fix where fix = 1 when $L = 2^Z$. Otherwise fix is equal to 0.

4.2 Verifying a Binary Balanced Cut-down Sequence

To verify a sequence is a binary balanced cut-down de Bruijn sequence the number of occurences of any length-M string on {0, 1} for $M \leq L$ as a substring of the sequence must be $\lfloor L/K^M \rfloor$ or $\lceil L/K^M \rceil$. To ensure efficiency an initial algorithm was implemented to count

the repetitions of each length-M string on {0, 1} that ran in O(n) time complexity. This algorithm iterates through the sequence $\mathcal{S}$ and computes base 10 value V from the current length-M string in $\mathcal{S}$. The value at the index V in an array holding these repetitions would be incremented by 1. After iterating through each length-M string the balance of the array is checked (no two substring repetitions may vary by at most 1). If the sequence is not balanced it is not a balanced cut-down sequence.

However, due to the exponential rate of length-M strings on {0, 1} needing to be checked this method becomes unfeasable quickly. Once $2^M \geq L$, for a sequence to be balanced the repetitions of each length-M string must be 0 or 1. Because of this property, when $2^M \geq L$ the verifier switches to a new method of iterating through the sequence $\mathcal{S}$ and checking if the current length-M sequence is repeated anywhere else in $\mathcal{S}$. If so, it is not a balanced cut-down de Bruijn sequence.

5 Conclusion

De Bruijn sequences have many applications such as the lock system problem [1] or the de Bruijn card trick problem [8]. However, the applications of these sequences are restricted because their lengths are limited to K^N where, K is the size of the sequence's alphabet and N is its order. For example, what if a member in the audience would like to see the magician do the de Bruijn card trick on a 52 card deck rather than the arbitrary 32. Luckily for the magician, variants of the de Bruijn sequence exist such as the cut-down de Bruijn sequence of which its length can be arbitrary while maintaining certain de Bruijn properties needed to perform the card trick. However, from the arbitrary length sequences discussed in this paper, the balanced cut-down de Bruijn sequence proved to be the most de Bruijn-like therefore, opening it up to greater applications.

References

[1] H. Fredricksen, "A survey of full length nonlinear shift register cycle algorithms," *SIAM review*, vol. 24, no. 2, pp. 195–221, 1982.

[2] S. W. Golomb, *Shift register sequences: secure and limited-access code generators, efficiency code generators, prescribed property generators, mathematical models.* World Scientific, 2017.

[3] J. Sawada, "De Bruijn sequence and universal cycle constructions." [Online]. Available: http://debruijnsequence.org/

[4] A. Lempel, "On a homomorphism of the de bruijn graph and its applications to the design of feedback shift registers," *IEEE Transactions on Computers*, vol. 19, no. 12, pp. 1204–1209, dec 1970.

[5] B. Cameron, A. Gündogan, and J. Sawada, "Cut-down de bruijn sequences," 2021.

[6] D. Gabric, Š. Holub, and J. Shallit, "Generalized de Bruijn words and the state complexity of conjugate sets," in *International Conference on Descriptional Complexity of Formal Systems*. Springer, 2019, pp. 137–146.

[7] A. Nellore and R. Ward, "Arbitrary-length analogs to de Bruijn sequences," *arXiv preprint arXiv:2108.07759*, 2021.

[8] P. Diaconis and R. Graham, "Magical mathematics," in *Magical Mathematics*. Princeton University Press, 2011.

Appendix A Binary Balanced Cut-down de Bruijn Sequence Constructor in C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <stdbool.h>

//Accounts for a size 1000 pkl sequence along with a size j wrap around to detect substrings
#define MAX_SIZE 1015

//Use for controlling execution in the for loop
bool exec = false;

//d_i digits for extension
int LbaseTwo[100];

//Length of necklace
int necklaceLength;

//Necklace characters
int a[MAX_SIZE];

//Lambda values for Lempel's lift
int Lambda0[MAX_SIZE];
int Lambda1[MAX_SIZE];

//Lambda modified strings for joining
int join0[MAX_SIZE];
int join1[MAX_SIZE];

//Function that computes the various substring functionalities of this program
//neck = the string being checked for substrings of subNeck,
//j = length of the substring, length of string being checked
int subIndex (int * neck, int * subNeck, int j, int l){
    bool substring = true;

    //Length of the necklace considering the wrap around
    int cyclicLength = l + j - 1;

    int tempNeck[MAX_SIZE];
    int cycleCount = 0;

    //Convert string into a cyclic string
    for(int i = 0; i < cyclicLength; i++){
        //Has reached the "end" of the necklace -- loop back to front
        if(cycleCount == l){
            cycleCount = 0;
        }
        tempNeck[i] = neck[cycleCount];
        cycleCount++;
    }

    //Check for substrings
    for(int i = 0; i < l; i++){
        int subIndex = 1;
        for(int n = i; n < i + j; n++){
            if(tempNeck[n] != subNeck[subIndex]){
                substring = false;
                break;
            }
            subIndex++;
        }
        if(substring == true){
```

```
                return i;
            }
            substring = true;
        }
        return -1;
    }

    //Caclulates Lambda0 and Lambda1
    void calculateLambda(int subscript){
        for(int z = 0; z < necklaceLength; z++){
            int neckSum = 0;
            for(int k = 0; k < z + 1; k++){
                if(a[k] == 1){
                    neckSum++;
                }
            }
            if(subscript == 0){
                Lambda0[z] = neckSum % 2;
            }
            else{
                neckSum++;
                Lambda1[z] = neckSum % 2;
            }
        }
    }

    //Sets a to Lambda0 concatenated with the complement of Lambda0
    void oddAlpha(){
        bool secondIter = false;
        int compNeck[MAX_SIZE];
        int count = 0;
        //set necklace a to Lambda0
        for(int i = 0; i < necklaceLength * 2; i++){
            //place complement of lambda0 into a temporary array
            if (secondIter == false){
                if(Lambda0[i] == 0){
                    compNeck[i] = 1;
                }
                else{
                    compNeck[i] = 0;
                }
                //Add the initial Lamba0 to alpha
                a[i] = Lambda0[i];
            }
            if(count == necklaceLength){
                count = 0;
                secondIter = true;
            }
            //Add the complement of lambda to alpha
            if(secondIter == true){
                a[i] = compNeck[count];
            }
            count++;
        }
    }

    //Return 0 for even weight and 1 for odd weight
    int Weight(){
        int num1s = 0;

        //calculate number of 1's
        for(int i = 0; i < necklaceLength; i++){
            if(a[i] == 1){
                num1s++;
            }
        }
        //p = 1 -- alpha has odd weight
        if((num1s % 2) != 0){
            return 1;
        }
        //alpha has an even weight
        return 0;
    }
```

```
//Modify the Lambda strings into a simpler form to join
void modifyString(int index, int strId){
    for(int i = 0; i < necklaceLength; i++){
        if(strId == 0){
            join0[i] = Lambda0[index];
        }
        else{
            join1[i] = Lambda1[index];
        }
        index++;
        if(index == necklaceLength){
            index = 0;
        }
    }
}

//Join strings Lambda0 and Lambda1 and assign the result to alpha
void joinAt(int index0, int index1){
    modifyString(index0, 0);
    modifyString(index1, 1);

    //Assign alpha to the joining of Lambda0 and Lambda1
    for(int i = 0; i < necklaceLength; i++){
        a[i] = join0[i];
    }
    for(int i = 0; i < necklaceLength; i++){
        a[i + necklaceLength] = join1[i];
    }
}

//insert 1 into k_j-1
void extend(int index){
    for(int i = necklaceLength; i > index; i--){
        a[i] = a[i - 1];
    }
    a[index] = 1;
    necklaceLength++;
}

//Determines if the length of the plk sequence is a result of 2^n where n > 0
int isPowOfTwo(int L){
    if (L == 1)
      return 1;

    // all other odd numbers are not powers of 2
    else if (L % 2 != 0 || L == 0)
      return 0;

    // recursive function call
    return isPowOfTwo(L / 2);
}

int main (int argc, char *argv[]){

    //Parameter validation
    if(argc != 2){
        printf("Incorrect command. Correct usage ./P2L L (Where 0 < L <= 1000)\n");
        exit(1);
    }

    //Assign L to the paramater
    int L = atoi(argv[1]);

    if(L < 1 || L > 1000){
        printf("L must be a positive integer less than or equal to 1000.\n");
        exit(1);
    }

    //Maintain desired P2L length - L is being modified
    int permL = L;

    //Calculate the ceiling of N
    int N = (int)ceil(log2(L));
```

```
int numBits = 0;
//Generating a base 2 representation of L
while(L != 0){
    LbaseTwo[numBits] = L % 2;
    L = L / 2;
    numBits++;
}

//Initialize alpha to the string "1"
a[0] = 1;
necklaceLength = 1;

int fix = 0;

//If L = 2, 4, 8, 16, 32, ... iterate an extra time through the loop
if(isPowOfTwo(permL) == 1){
    fix = 1;
}

for(int j = 1; j < N + fix; j++){

    //Construct Lempel's Lift
    calculateLambda(0);
    calculateLambda(1);

    //p = 1 -- alpha has odd weight
    if(Weight() == 1){
        oddAlpha();
        exec = true;
    }

    //if 1_j-1 is a substring of alpha
    int subNeck0[100];
    for(int n = 1; n < j; n++){
        subNeck0[n] = 1;
    }

    if(subIndex(a, subNeck0, j - 1, necklaceLength) != -1 && exec == false){
        int subStr[100];

        for(int n = 1; n < j; n++){
            subStr[n] = (n - 1) % 2;
        }

        int index0 = subIndex(Lambda0, subStr, j - 1, necklaceLength);
        int index1 = subIndex(Lambda1, subStr, j - 1, necklaceLength);

        joinAt(index0, index1);
        exec = true;
    }

    //If lambda0 and Lamba1 can be join at (0_j-2^++, k) or (k, 0_j-2^++)
    int subNeck1[100];
    int n;
    for(n = 1; n < j - 1; n++){
        subNeck1[n] = (n - 1) % 2;

    }
    //Adds on k % 2 after prefix
    subNeck1[n] = 0;

    //If lambda0 and Lamba1 can be join at (0_j-2^++, k)
    if((subIndex(Lambda0, subNeck1, j - 1, necklaceLength) != -1 &&
    subIndex(Lambda1, subNeck1, j - 1, necklaceLength) != -1) && exec == false){

        int index0 = subIndex(Lambda0, subNeck1, j - 1, necklaceLength);
        int index1 = subIndex(Lambda1, subNeck1, j - 1, necklaceLength);

        joinAt(index0, index1);
        exec = true;
    }

    //Adds on k % 2 as prefix
    subNeck1[1] = 0;
```

```
            for(n = 2; n < j; n++){
                subNeck1[n] = (n - 2) % 2;
            }

            //If lambda0 and Lamba1 can be join at (k, 0_j-2^++)
            if((subIndex(Lambda0, subNeck1, j - 1, necklaceLength) != -1 &&
            subIndex(Lambda1, subNeck1, j - 1, necklaceLength) != -1) && exec == false){

                int index0 = subIndex(Lambda0, subNeck1, j - 1, necklaceLength);
                int index1 = subIndex(Lambda1, subNeck1, j - 1, necklaceLength);

                joinAt(index0, index1);
                exec = true;
            }

            //Set alpha to joining Lambda0 and Lambda1 at 0_j-2^++
            if(exec == false){
                int subNeck2[100];
                for(n = 1; n < j - 1; n++){
                    subNeck2[n] = (n - 1) % 2;
                }
                int index0 = subIndex(Lambda0, subNeck2, j - 2, necklaceLength);
                int index1 = subIndex(Lambda1, subNeck2, j - 2, necklaceLength);

                joinAt(index0, index1);
            }
            exec = false;
            necklaceLength *= 2;

            //If d_j = 1 then extend alpha at K_j-1
            if(LbaseTwo[numBits - j - 1] == 1){
                extend(subIndex(a, subNeck0, j - 1, necklaceLength));
            }
        }

        //Display pk1 sequence
        printf("a = ");
        for(int i = 0; i < necklaceLength; i++){
            printf("%d", a[i]);
        }
        printf("\n");

        return 0;
    }
```

Appendix B Binary Balanced Cut-down de Bruijn Sequence Verifier in C

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <math.h>
    #include <stdbool.h>
    #include <time.h>

    //Accounts for a size 1000 p2l sequence along with a size L wrap around
    #define MAX_SIZE 2030

    int s[MAX_SIZE];
    int initSeq[MAX_SIZE];
    int tempSeq[MAX_SIZE];
    int initLength;
    char initStr[MAX_SIZE];


    //Calculate the base 10 representation of bits between left and right bounds
    void computeReps(int left, int right){
        int sum = 0;
        for (int i = left; i <= right; i++) {
```

```
            if(tempSeq[i] == 1){
                sum+= pow(2, (right - i - 1));
            }
        }
        s[sum] = s[sum] + 1;
    }

    void modifyString(int m, int l){

        //Copy the original sequence to our m-version
        for(int i = 0; i < l; i++){
            tempSeq[i] = initSeq[i];
        }

        //Account for wrap-around
        for(int i = 0; i < m; i++){
           tempSeq[l + i] = initSeq[i];
        }
    }

    int subIndex (int * subNeck, int m){
        int totalSubstring = 0;
        bool substring = true;

        //Check for substrings
        for(int i = 0; i < initLength; i++){
            int subIndex = 0;
            for(int n = i; n < i + m; n++){
                if(tempSeq[n] != subNeck[subIndex]){
                    substring = false;
                    break;
                }
                subIndex++;
            }
            if(substring == true){
                totalSubstring++;
            }
            substring = true;
        }
        return totalSubstring;
    }

    void betweenOne(int maxSeq, int m){
        int max = s[0];
        int min = s[0];

        for(int i = 1; i < maxSeq; i++){
            if(s[i] < min){
                min = s[i];
            }
            if(s[i] > max){
                max = s[i];
            }
        }
        if((max - min) > 1){
            printf("\n\nNot a PLK sequence. Occurences of length-%d
            strings vary from at most %d\n\n", m, max - min);
            exit(1);
        }
    }

    int main(int argc, char *argv[]){

        printf("Enter PLK sequence: ");
        scanf("%s", &initStr);

        //Calculate length of pkl sequence
        initLength = strlen(initStr);

        //Convert string to array of bits
        for(int i = 0; i < initLength; i++){
            initSeq[i] = (int) initStr[i] - 48;
        }
```

```
        int m = 0;
        for(int i = 1; i < initLength + 1; i++){
            m = i;
            int maxSeq = pow(2, m);

            if(maxSeq >= initLength){
                //Not feasable to continue generating all strings of length-m
                break;
            }

            //init array for new set of m-length strings
            for(int i = 0; i < maxSeq; i++){
                s[i] = 0;
            }

            //Implement cyclic property
            modifyString(m, initLength);

            for(int i = 0; i < initLength; i++){
                computeReps(i, i + m);
            }

            betweenOne(maxSeq, m);
        }

        //Now check the sequence to see if any substring is repeated > 1
        int left = 0;

        for(int i = m; i < initLength + 1; i++){
            //Implement cyclic property
            modifyString(i, initLength);

            for(int n = 0; n < initLength; n++){
                int mString[MAX_SIZE];
                for(int k = 0; k < i; k++){
                    mString[k] = tempSeq[n + k];
                }
                if(subIndex(mString, i) > 2){
                    printf("\n\nNot a PLK sequence the length-%d string: ", i);
                        for(int k = 0; k < i; k++){
                            printf("%d", mString[k]);
                        }
                        printf(" occurs > 1 while k^m > L.\n\n\n");
                        exit(1);
                }
                left++;
            }
        }
        printf("\nThis is a valid PLK sequence.\n");
        return 0;
    }
```