Finding the Largest Fixed-Content Necklace

Matthew Ng

Joseph Sawada, University of Guelph

CIS4910

April 2022

Abstract

A necklace is the lexicographically least element in a set of all rotations of a string. Fixed-content refers to a list of symbols, where the number of occurrences of each symbol in the list is the same as in a particular string. In this paper, we present an algorithm that finds the lexicographically largest necklace of a content C in $O(n^2)$ time.

1 Introduction

A *necklace* is the lexicographically least element in a set of all rotations of a string. A necklace is *periodic* when two strings α, β are such that $\alpha = \beta^t$ for some $t > 1$. *Aperiodic* necklaces are referred to as a *Lyndon word* [5]. A necklace with *fixed-content* refers to a necklace with a specified content of symbols in a k-ary alphabet represented by a list of all such symbols. The goal of this paper is to present an efficient algorithm that finds the largest fixed-content necklace. A necklace with *fixed-density* refers to a binary necklace with a specified quantity of 1s.

The primary motivation behind this algorithm is to solve the overarching problem of efficiently ranking and unranking fixed-content necklaces and Lyndon words. Past research into this problem goes back to the first polynomial-time ranking and unranking algorithms presented in 2014 by Kociumaka, Radoszewski, and Rytter [3]. This algorithm was further extended upon by Hartman and Sawada, creating an algorithm for ranking and unranking fixed-density necklaces and Lyndon words in polynomial-time [2]. As a major step to this ranking and unranking algorithm, an algorithm that generates the largest fixed-density necklace in polynomial time was found in 2017. In order to generalize the binary ranking algorithm into the k-ary case, we similarly find an algorithm that finds the largest fixed-content necklace of length n to aid in finding an algorithm for ranking and unranking necklaces and Lyndon words.

Example 1. Listing of all necklaces with content $[0, 0, 1, 2, 2, 2]$

- **021022**
- 020212
- 012022
- 002221
- 002122
- 020221
- 012202
- 010222
- 002212
- 001222

Given the aforementioned content, the algorithm will return 021022 as the largest fixed-content necklace.

Interest in ranking and unranking fixed-content necklaces and Lyndon words was catalyzed by an algorithm by Cameron, Gündoğan, and Sawada on the efficient construction of binary cut-down de Bruijn sequences. Construction of the cut-down de Bruijn sequences made use of a recently discovered algorithm by Hartman and Sawada that ranks and unranks fixed-density necklaces and Lyndon words in polynomial time [6]. Generalizing the algorithm to work with a k-ary alphabet requires a polynomial-time algorithm for ranking fixed-content Lyndon words [1].

Section 2 of this paper will provide background and notation for the algorithm, introducing important aspects to the algorithm, such as the notion of multi-way partitioning. In Section 3, we present our current progress on a recursive algorithm that generates the largest fixed-content necklace in $O(n^2)$ time, as well as an $O(n)$ algorithm that performs multi-way partitioning of content C such that the lexicographically least string in the partition is maximized. In Section 4, we discuss our results and future plans for this algorithm. The appendix features an implementation of the algorithm in Python. All algorithms runtimes are to be examined using the word-RAM model.

2 Background and Notation

Let $\alpha = \alpha_1\alpha_2\ldots\alpha_n, \beta = \beta_1\beta_2\ldots\beta_m$ be strings. α is considered lexicographically lesser than β if either [2]:

1. $\alpha_1\alpha_2\ldots\alpha_{j-1} = \beta_1\beta_2\ldots\beta_{j-1}$ for some $j \leq n \leq m, j \geq 1$, or
2. $\alpha_1\alpha_2\ldots\alpha_n = \beta_1\beta_2\ldots\beta_m$, where $n < m$.

We define the *rank* of a combinatorial object in some list of all distinct combinatorial objects as its position in the list. For example, given a list of combinatorial objects $\alpha_1, \alpha_2, \ldots, \alpha_m$ the *rank* of α_i is i. The process of ascertaining the rank of a certain object is called *ranking*. The process of retrieving the object given a certain rank is called *unranking* [7].

Let C be a list of all lexicographical content of an alphabet of k size in lexicographically descending order.

As a precursor to a lemma involving the proper distribution of content, we introduce the general problem of *multi-way number partitioning*. Given a multiset S of n positive integers and k subsets, the multi-way number partitioning problem seeks to optimally distribute the content of S in ways that make up three objective functions [8]:

1. Minimize the largest of the k subset sums.
2. Maximize the smallest of the k subset sums.
3. Minimize the difference between the largest and smallest of the k subset sums.

Example 2. Results of the objective functions.
Given $S = \{13, 9, 9, 6, 6, 6\}$ and $k = 3$:

1. Minimize the largest of the k subset sums.

$$\{13\}, \{9, 9\}, \{6, 6, 6\}$$

2. Maximize the smallest of the k subset sums.

$$\{13, 6\}, \{9, 6\}, \{9, 6\}$$

3. Minimize the difference between the largest and smallest of the k subset sums

$$\{13\}, \{9, 9\}, \{6, 6, 6\}$$

Note that while for this particular input, objective functions 1 and 2 return the same result, they are not equivalent for $k > 2$ [4].

3 Finding the largest fixed-content necklace

In this section we devise a recursive algorithm that finds the lexicographically largest necklace denoted as $LargestNecklace$ of length n given fixed content C of an alphabet of size k. We denote this algorithm by function $LargestNeck(C)$.

The basic idea behind this algorithm is to recursively multiway-partition the content into each of the substrings that are separated by a 0 in what will be the resultant largest necklace and continually made into a simplified content until it hits a base case such that finding the ordering of the necklace is trivial. Then, the moving up the recursive stack, the resultant string will be expanded by replacing each simplified content in the string with the generated substring from the multiway partitioning.

Lemma 1. *LargestNecklace takes the form* $a = 0w_1 0w_2 \cdots 0w_z$ *where* w_i *is a word of length* ≤ 0*, and* $content(S_w) = C$*.*
Proof. Suppose $LargestNeck(C) = w_1 0w_2 0 \cdots w_z 0$. Since w_i does not contain any 0, $w_i > 0$. Thus, shifting the string left by the length of w_1, or any length such that 0 is the leading symbol garners a smaller string which is a contradiction. Therefore, *LargestNecklace* must be in the form $0w_1 0w_2 \cdots 0w_z$.

Lemma 2. *Let* z *be the number of 0s in* C *which can be partitioned into a list of* z *strings in* $O(n)$ *time in a similar manner to objective function 2 in Section 2, such that the lexicographically least string is maximized.*
Note. Let $T(C)$ be the set of all unique sets S_i of n_0 strings, where $content(S_i) = \mathrm{C}$, where $content(S)$ returns an enumerated list in lexicographically descending order of the combined content each string in S. Let Z_i be the lexicographically smallest string in S_i, where Z_A in S_A is the lexicographically largest for any Z_i in any list S_i in $T(C)$.

The multiway partitioning goes as follows. A list of z many lists is created to serve as each partition of content. Starting from the lexicographically largest symbol, it is evenly distributed from the top down to the bottom, wrapping around to the start of the list until all of such symbols are exhausted. The list that the symbol is stopped on is considered *differentiated*. If the differentiated list is the last or second last list, then all of the following content up until 0 non-inclusive is placed into the last list in reverse lexicographical order. Otherwise, the same process is started, where the next lexicographically largest symbol will be distributed starting from the next list after the differentiated list. Since there is only one constant time operation made per symbol, this will run in $O(n)$ time.

A proof would likely entail a proof by contradiction, supposing Z_A is not the lexicographically largest of its kind in $T(C)$.

Example 3. Multiway partitioning of a given content.
Let C = [5,5,5,5,5,5,5,5,4,4,4,4,4,4,4,4,3,3,3,3,2,2,2,1,1,1,1,0,0,0,0,0,0,0].

$$[\,], [\,], [\,], [\,], [\,], [\,]$$

$z = 6$ lists are created, serving as the partitions of content.

$$[5, 5], [5], [5], [5], [5], [5]$$

All 5s are evenly distributed, wrapping around from the top. The first list is now considered differentiated. Distribution of content will now start from the second list.

$$[5, 5], [5, 4, 4], [5, 4, 4], [5, 4, 4], [5, 4], [5, 4]$$

The next lexicographically largest symbol is distributed through the lists. Notice how the wraparound starts at the list directly after the differentiated list.

$$[5, 5], [5, 4, 4], [5, 4, 4], [5, 4, 4], [5, 4, 3, 3], [5, 4, 3]$$

Going down the list, the penultimate list is now considered differentiated.

$$[5, 5], [5, 4, 4], [5, 4, 4], [5, 4, 4], [5, 4, 3, 3], [5, 4, 3, 2, 2, 2, 1, 1, 1, 1]$$

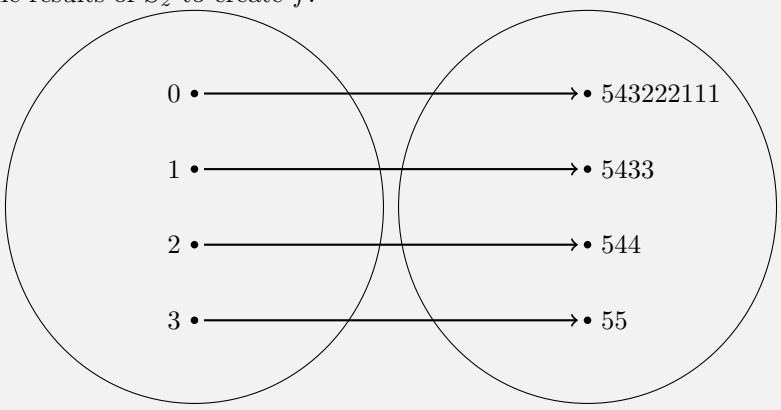
All remaining non-zero content is then distributed into the last list. With this, we get strings $55, 544, 544, 5433, 5432221111$, where the lexicographically least string 5432221111 is maximized.

Definition 1. *A function MultiwayPartition(C, z) following the properties from Lemma 2 will return a list of tuples S_z, where each tuple consists of its lexicographical rank in the resulting list of strings and the string itself. In $LargestNeck(C)$, a bijective mapping is created using the information in S_z defined as $f : i \to w_i$ with input of a new content value to a resultant string.*
A new content, C' for the next recursive call, $LargestNecklaces(C')$ can be formed using S_z.

Example 4. S_z given the results of Example 2.
Given the list of strings $[55, 544, 544, 5433, 543222111]$, we are given 4 unique elements. S_z is as such:

$$[(0, 543222111), (1, 5433), (2, 544), (3, 55)]$$

We can use the results of S_z to create f:



Which we can also represent as $\{0 : 543222111, 1 : 5433, 2 : 544, 3 : 55\}$. With S_z, we are also given a new content $C' = [0, 1, 1, 2, 3]$.

Lemma 3. Following Lemma 1, necklace $w_1 w_2 \cdots w_z$ can be produced by recursively calling $LargestNeck(C')$ from the results of MultiwayPartition(C, z) and decoding the recursively generated necklace using f.
Note: A proof of this would most likely use the results from Lemma 2's proof, but must also extend on it to ensure that objective function 2 of the multiway partitioning problem actually maximizes the necklace. Additionally, since z many constant time operations will be made to f and $z \leq n$, the recursive call retains $O(n)$ time.

Lemma 4. *Let $0 < z \leq \frac{n}{2}$. Then*

$$LargestNeck(C) = 0w_1 0w_2 \cdots 0w_z$$

Let $M = LargestNeck(C')$. Then $w_1 w_2 \cdots w_z = f[M[0]] \cdot f[M[1]] \cdots f[M[z-1]]$.

Note: The proof of this would likely simply refer back to the Lemma 2's proof.

Lemma 5. *Let $0 < \frac{n}{2} < z$. Then*

$$LargestNeck(C) = 0w_1 0w_2 \cdots 0w_j$$

Where j is equal to the length of the content minus z. Let $M = LargestNeck(C')$. Then $w_1 w_2 \cdots w_z = f[M[0]] \cdot f[M[1]] \cdots f[M[k-1]]$.
Note: MultiwayPartition(C, z) holds the special case for which each prefix of strings in S_z is evenly distributed $z - k - 1$ 0s. This is an optimization for the previous lemma for the case of when 0 makes up the majority of the content.

A proof of this would likely work on the basis of the intuitive observation of prioritizing the padding the lexicographically largest symbols with zeroes would lead to the objective function of maximizing the lexicographically least string.

Example 5. MultiwayPartition(C, z) given content of majority 0s.
Let $C = [3, 3, 3, 2, 2, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$.

$$[\,], [\,], [\,], [\,], [\,], [\,], [\,]$$

$n - z = 7$ lists are created to serve as the partitions for the new strings.

$$[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0], [0, 0], [0, 0]$$

$z - k - 10s$ are evenly appended to each list.

$$[0, 0, 0, 3], [0, 0, 0, 3], [0, 0, 0, 3], [0, 0, 0, 2], [0, 0, 2], [0, 0, 1], [0, 0, 1]$$

All non-zero content is added in lexicographically descending order.

Lemma 6. *Let $z = 1$. Then*

$$LargestNeck(C) = 0 \cdot C[n-1] \cdot C[n-2] \cdots C[1]$$

Note: The proof should be relatively simple. The intuitive observation should go as follows: the largest string that can be made from any content is a string of the all content in lexicographically descending order. The next observation is that all necklaces of a content where $z \geq 1$ will begin with a 0 through Lemma 1. Therefore, a necklace that begins with a 0 then features the rest of the content in reverse lexicographical order will be the largest fixed-content necklace.

Lemma 7. *Let $k = 1$. Then*

$$LargestNeck(C) = 0^n$$

Note: The proof is obvious. There is only one possible string given this content, the smallest rotation is that string.

This results in the following recurrence for $LargestNeck(C)$:

$$LargestNeck(C) = \begin{cases} 0w_1 0w_2 \cdots 0w_z & if\ 0 < z \leq \frac{n}{2} \\ 0s_1 0s_2 \cdots 0s_j & if\ 0 < \frac{n}{2} < z \\ 0 \cdot C[n-1] \cdot C[n-2] \cdots C[1] & if\ z = 1 \\ 0^n & if\ k = 1 \end{cases}$$

where $\{w_1, w_2, \ldots, w_z\} = MultiwayPartition(C, z)$ and $\{s_1, s_2, \ldots, s_j\} = MultiwayPartition(C, z)$.

Using Lemmas 1 and 3, we can deduce that $LargestNeck(C)$ will run in $O(n^2)$ time. Observing that each recursive content mapping will shorten the content by at least 1 character from Lemma 4, we can deduce recursive calls are made to the order of n times where each call runs in $O(n)$ time from Lemma 1 and thus, $LargestNeck(C)$ has a time complexity of $O(n^2)$.

Theorem 1. *The largest necklace of length n and content C can be computed in $O(n^2)$.*

We conclude this section with the pseudocode for the functions $MultiwayPartition(C, z)$ and $LargestNeck(C)$.

Algorithm 1 Partitioning Algorithm

```
function MultiwayPartition(C, z)
    S_z ← [], d ← 0, n ← len(C)
    if z ≤ n/2 then
        for i ← 0 to z − 1 do
            append [] to S_z
        for i ← n − 1 to 0 do
            if C[i] = 0 then break
            append C[i] to S_z[i mod (z − d)]
            if i < n and C[i]! = C[i + 1] then
                d ← i mod (z − d)
    else
        j ← n − z
        for i ← 0 to j do
            append [] to S_z
        for i ← 0 to z − k − 2 do
            append 0 to S_z[i mod len(S_z)]
        for i ← 0 to n − 1 do
            if C[i] = 0 then break
            append C[i] to S_z[i]
    temp ← []
    for i ← len(S_z) − 1 to 0 do
        if len(temp)! = 0 or temp[len(temp) − 1]! = S_z[i] then
            append S_z[i] to temp
     j ← 0
    for i ← 0 to len(S_z) − 1 do
        if S_z[i]! = temp[j] then
            j ← j + 1
        S_z[i] ← (j, S_z[i])
    return S_z
```

From Definition 1, we get pseudocode for the function MultiwayPartition(C, z). Note the two cases from Lemmas 6 and 7. This algorithm runs in $O(n)$ time.

Algorithm 2 Largest Necklace Algorithm

```
function LargestNeck(C)
    if z = 1 then return 0 · C[n − 1] · C[n − 2] · · · C[1]
    if k = 1 then return 0^n
    S_z ← MultiwayPartition(C)
    f ← map(), C' = []
    for i ← 0 to len(S_z) − 1 do
        f[S_z[i][0]] = S_z[i][1]
        if len(C') = 0 or C'[len(C') − 1]! = S_z[i][0] then
            append S_z[i][0] to C'
    LN ← LargestNeck(C')
    return 0 · f[LN[0]] · 0 · f[LN[1]] · · · 0 · f[LN[len(LN) − 1]]
```

Following Theorem 1, we arrive at pseudocode for the recursive largest fixed-content necklace construction algorithm which runs in $O(n^2)$ time.

Example 6. Generating the largest fixed-content necklace
Let $C = [2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 0, 0, 0, 0]$,
Since $0 < z \leq \frac{n}{2}$, we generate $z = 4$ many lists in $MultiwayPartition(C, z)$ which forms into this content:

$$[2, 2], [2, 2], [2, 1, 1, 1], [2, 1, 1, 1]$$

which results in $S_z = [(0, 2111), (0, 2111), (1, 22), (1, 22)], f = \{0 : 2111, 1 : 22\}, C' = \{1, 1, 0, 0\}$. Making the recursive call, $LargestNeck(C')$, we arrive at the previous case. Let R be the content and x represent the number of 0s in R in this recursive call.
We call $MultiwayPartition(R, x)$, forming the content:

$$[1], [1]$$

resulting in $T_z = [(0, 1), (0, 1)], g = \{0 : 1\}, R' = \{0, 0\}$. Making a new recursive call with the new content, we hit the basecase of $k = 0$, so we return the string 00.
Moving up the recursive stack, 00 is decoded into

$$0101$$

using g and returned up the recursive stack. 0101 decodes to

$$0211102202111022$$

which is the resultant LargestNecklace of content C.

4 Future Work and Closing Remarks

Following Hartman and Sawada's work on generating largest fixed-density necklaces in $O(n^2)$ time [6], in this paper, we have extended upon the functionality of the algorithm to generate fixed-content necklaces of a k-ary alphabet in $O(n^2)$ time. With this, there is still much work to be done with this algorithm and the overarching goal of being able to rank and unrank necklaces and Lyndon words in polynomial time.

There are two main obvious next steps to this research. The first is to devise proofs for the remaining lemmas that support the algorithm. The other is to extend this generation algorithm to Lyndon words. We then want to solve the problem of finding the largest necklace of a given content less than or equal to a necklace of the same content in a similar manner to previous work by Hartman and Sawada in the pursuit of a ranking and unranking algorithm [6].

In addition to this, we conjecture that the multi-way partitioning of the content, such that the smallest subset sub is maximized in objective function 2 can be helpful for certain cases for the cut-down de Bruijn sequence generation algorithm by Cameron, Gündoğan, and Sawada for generalization to a k-ary alphabet [1].

References

[1] Ben Cameron, Aysu Gündoğan, and Joe Sawada. Cut-down de Bruijn sequences. 2021.

[2] Patrick Hartman and Joe Sawada. Ranking and unranking fixed-density necklaces and Lyndon words. *Theor. Comput. Sci.*, 791:36–47, 2019.

[3] Tomasz Kociumaka, Jakub Radoszewski, and Wojciech Rytter. Efficient ranking of Lyndon words and decoding lexicographically minimal de Bruijn sequence. *SIAM J. Discret. Math.*, 30(4):2027–2046, 2016.

[4] Richard E. Korf. Objective functions for multi-way number partitioning. In Ariel Felner and Nathan R. Sturtevant, editors, *Proceedings of the Third Annual Symposium on Combinatorial Search, SOCS 2010, Stone Mountain, Atlanta, Georgia, USA, July 8-10, 2010*. AAAI Press, 2010.

[5] Joe Sawada. A fast algorithm to generate necklaces with fixed content. *Theor. Comput. Sci.*, 301(1-3):477–489, 2003.

[6] Joe Sawada and Patrick Hartman. Finding the largest fixed-density necklace and Lyndon word. *Inf. Process. Lett.*, 125:15–19, 2017.

[7] Joe Sawada and Aaron Williams. Practical algorithms to rank necklaces, Lyndon words, and de Bruijn sequences. *J. Discrete Algorithms*, 43:95–110, 2017.

[8] Ethan L. Schreiber, Richard E. Korf, and Michael D. Moffitt. Optimal multi-way number partitioning. *J. ACM*, 65(4):24:1–24:61, 2018.

Appendix A: Python Partitioning Code

```
from collections import defaultdict

def createPartition(inputNums):
    # content inserted as a list
    diff = 0
    zeroes = int(inputNums[0])
    # partitions of zeroes are generated
    patterns = [["0"] for i in range(zeroes)]
    n = sum(inputNums)

    # two cases for multiwayPartition, if zeroes makes up most of list and otherwise
    if zeroes <= n/2:
        for i in range(len(inputNums) - 1, 0, -1):
            keeptrack = 0
            # place numbers between differentiated number and end of list
            for j in range(int(inputNums[i])):
                j%(len(patterns) - diff) + diff
                patterns[j%(len(patterns) - diff) + diff].append(str(i))
                keeptrack = j%(len(patterns) - diff ) + diff
            diff = keeptrack + 1 if (keeptrack + 1) < len(patterns) else diff
    else:
        # optimization for multiwayPartition
        j = n - zeroes
        patterns = [["0"] for i in range(j)]
        for i in range(zeroes - len(patterns)):
            patterns[i%len(patterns)].append("0")
        k = 0
        for i in range(len(inputNums) - 1, 0, -1):
            for l in range(inputNums[i]):
                patterns[k].append(str(i))
                k += 1

    # maps each partition to amount of occurances
    tableOfPatterns = defaultdict(int)
    # maps the original pattern to its occurances
    sortedListOfPatterns = []
    for pattern in patterns:
        strPattern = ''.join(pattern)
        tableOfPatterns[strPattern] += 1
    for key in tableOfPatterns:
        sortedListOfPatterns.append(str(key))
    sortedListOfPatterns.sort()
    newContent = []
    # create new pattern
    for i in range(len(sortedListOfPatterns)):
        newContent.append(tableOfPatterns[sortedListOfPatterns[i]])
    mapOfContent = {}
    # create a mapping of new symbols to the constituent pattern
    for i in range(len(sortedListOfPatterns)):
        mapOfContent[i] = sortedListOfPatterns[i]

    return newContent, mapOfContent
```

```
def construct(necklace, mapOfContent):
    # split into all patterns
    patterns = ["" for i in range(len(necklace))]
    for i in range(len(patterns)):
        # substitute every subbed symbol into its constituent pattern at this point in
                                                                the recursion
        patterns[i] = str(mapOfContent[int(necklace[i])])
    resultNecklace = ""
    # reconstruct the necklace string
    for pattern in patterns:
        resultNecklace += str(pattern)
    return resultNecklace

def recurseNeck(content):
    # should return a hashmap containing a sorted list of the patterns and a map
                                                containing each pattern and the
                                                occurances
    newContent, mapOfContent = createPartition(content)
    # base case is if there is only one zero or if the only symbol is zero
    if len(newContent) == 1:
        newNecklace = "0" * newContent[0]
        return construct(newNecklace, mapOfContent)
    if newContent[0] == 1:
        newNecklace = "0"
        for i in range(len(newContent) - 1, 0, -1):
            for j in range(newContent[i]):
                newNecklace += str(i)
        return construct(newNecklace, mapOfContent)
    # return the necklace constructed when moving up call stack
    return construct(recurseNeck(newContent), mapOfContent)

def go():
    print("enter k: ")
    n = int(input())
    content = []
    for i in range(n):
        print("enter n for", i)
        curr = input()
        content.append(curr)
    if len(content) == 1:
        print('0'*content[0])
    else:
        observed = []
        for i in range(len(content)):
            for j in range(content[i]):
                observed.append(str(i))
        observed = observed[::-1]
        print("content:", observed)
        print(recurseNeck(content))

go()
```