Abstract. Lists of equivalence classes of words under rotation or rotation plus reversal (i.e., necklaces and bracelets) have many uses, and efficient algorithms for generating these lists exist. In combinatorial group theory elements of a group are typically written as words in the generators and their inverses, and necklaces and bracelets correspond to conjugacy classes and relators respectively. We present algorithms to generate lists of freely and cyclically reduced necklaces and bracelets in free groups. Experimental evidence suggests that these algorithms are CAT — that is, they run in constant amortized time.

Key words. necklace, bracelet, CAT algorithm, free group, reduced word, conjugacy class

AMS subject classifications. 05A05, 20E05, 20E45, 20F05

1. Introduction. Given an ordered alphabet of size $k$, a necklace of length $n$ is the lexicographically least element of an equivalence class of $k$-ary strings of length $n$ under rotation. A word is called a prenecklace if it is the prefix of some necklace. An aperiodic necklace is called a Lyndon word. A bracelet of length $n$ is the lexicographically least element of an equivalence class of $k$-ary strings of length $n$ under string rotation and string reversal.

For a fixed $k$ the number of necklaces (also Lyndon words, prenecklaces and bracelets) grows exponentially with the length. See, for example, [3, 13] for exact counts and bounds. So generating a complete list of the length $n$ necklaces takes exponential time, and our goal is an algorithm where the computation (the total amount of change to the data structures, not including any processing of the generated necklaces) is proportional to the number of necklaces generated. Such an algorithm is a constant amortized time, or CAT, algorithm.

In group theory, elements of a group can be represented by strings (or words) in the group’s generators and their inverses. Symbolic algebra systems such as GAP and MAGMA [1, 2] make sophisticated testing of large numbers of examples straightforward, and efficient algorithms for generating complete lists of words, up to some equivalence, are an important part of this. The extant enumeration algorithms do not take into account the group structure, and we demonstrate how they can be recast to address this. For necklaces this process is trivial, while for bracelets we need modify the reversal checking code materially.

The remainder of this paper is organized as follows. Section 2 gives some background material on necklaces and bracelets and on free groups and group presentations, and discusses the analogues of necklaces and bracelets in groups. Sections 3 and 4, respectively, describe our necklace and bracelet listing algorithms, and we discuss our results in Section 5. Appendix A describes the tests we performed on the running times of our algorithms.

2. Background. The first algorithm for generating necklaces, the FKM algorithm (due to Fredericksen, Kessler and Maiorana [6, 7]), was proved to be CAT in [11]. A simple recursive CAT algorithm to generate prenecklaces, necklaces and Lyndon words was given in [3], and it is this algorithm which forms the basis of our work.

*School of Information Technology and Electrical Engineering, The University of Queensland, Australia (uqcramsa@uq.edu.au).
Duval’s algorithm for factoring a string, of length \( n \), into Lyndon words [5] yields an algorithm for generating the necklace of the string in \( O(n) \) time. Thus a straightforward approach to generating bracelets is to generate the necklaces and to reject those where the necklace of the reversal is less than the necklace. However, this does not yield a CAT algorithm. The algorithm given in [13] is based on the recursive algorithm of [3] and maintains auxiliary data regarding the current prenecklace, using this to guide testing against its reversal and control the computation. The total amount of extra work, amortized over all bracelets, is constant, so this bracelet generating algorithm is CAT.

Given a set \( S \) of \( g \) symbols define the set \( S' = S \cup \{s^{-1} : s \in S\} \). The set of all words on \( S' \) is \( F_g \), the free group of rank \( g \). The group operation is concatenation, \( s^{-1} \) is read as the inverse of \( s \), and the empty word is the identity element. Words with no substrings of the form \( ss^{-1} \) or \( s^{-1}s \) are called freely reduced. Two words represent the same element of \( F_g \) if and only if they are identical after being freely reduced. (See [8, 9] for more details on combinatorial group theory.)

Let \( w = s_1 \cdots s_\ell \in F_g \). If \( s_1 \) and \( s_\ell \) are not inverses of each other, then \( w \) is cyclically reduced. Given an \( r \in S' \) then the word \( r^{-1} wr \) is the conjugate of \( w \) by \( r \), denoted \( w^r \). If \( r = s_1 \) (resp., \( s_\ell^{-1} \)) and the substring \( r^{-1}s_1 \) (resp., \( s_\ell^r \)) is canceled from \( w^r \) then the resulting word is a rotation of \( w \) by one position. If \( r = s_1 = s_\ell^{-1} \) then canceling \( r^{-1}s_1 \) and \( s_\ell^r \) performs a cyclic reduction step and reduces the length of \( w \) by two. Repeated conjugation of a word may render it cyclically reduced or freely reduced, rotate it arbitrarily, or increase its length arbitrarily.

Conjugation partitions the words in \( F_g \) into conjugacy classes. Given an order on \( S' \), we take as class representatives the lexicographically least element of the freely and cyclically reduced words in the class. So, in the context of \( F_g \), listing the freely and cyclically reduced necklaces of length \( \ell \) is equivalent to listing the conjugacy classes whose shortest words have length \( \ell \). A word which is both freely reduced and cyclically reduced will be called simply reduced.

Reversing a word \( w \) is not meaningful in \( F_g \). However reversing \( w \) and then replacing each of its elements by its inverse gives \( w^{-1} \) (i.e., freely reducing \( ww^{-1} \) or \( w^{-1}w \) results in the empty word). Given a set of words \( R \) in \( F_g \), the normal closure \( N \) of \( R \) in \( F_g \) is the set of all words which are concatenations of conjugates of the words in \( R \) and their inverses. Groups are often described as quotient groups of free groups and \( N \) is a normal subgroup of \( F_g \), so the quotient \( F_g/N \) describes some group \( G \). (Formally, there is a homomorphism from \( F_g \) onto \( G \) with kernal \( N \), see [8, 9].)

The pair \( (S, R) \) is a presentation for \( G \), written as \( G = \langle S : R \rangle \). The elements of \( S \) are the generators of \( G \). The words in \( R \) are equal to the identity in \( G \) and are called relators. So, in the context of \( F_g \), listing the reduced bracelets of length \( \ell \) is equivalent to listing equivalence classes of possible relators of length \( \ell \) in a presentation.

Enumerating reduced necklaces and bracelets in groups is equivalent to enumerating general necklaces and bracelets with forbidden substrings. An efficient algorithm exists to enumerate necklaces with a forbidden substring [12], however the analysis therein assumes that the substring has length at least three. For substrings of length one or two, [12] notes that “trivial algorithms can be developed”. In our case, we simply test each potential addition to the current prenecklace, and skip those which cannot yield a reduced necklace.

In the remainder of this paper, unless explicitly stated otherwise, we are always working in the free group \( F_g \). The number of group generators will be denoted by \( g \) and the word length by \( \ell \) (both assumed positive), with the set of possible symbols in our words having size \( k = 2g \). From [10, Theorems 1.1 & 14.2] we have the following
result.

**Theorem 2.1.** The number of reduced words of length \( \ell \) in \( F_g \) is equal to

\[
C(g, \ell) = \begin{cases} 
(2g-1)^\ell + 1, & \text{if } \ell \text{ is odd}; \\
(2g-1)^\ell + 2g - 1, & \text{if } \ell \text{ is even}. 
\end{cases}
\]

Let \( \phi \) denote the Euler totient function. Then the number of reduced necklaces of length \( \ell \) in \( F_g \) is equal to

\[
CC(g, \ell) = \frac{1}{\ell} \sum_{d|\ell} \phi(d) C(g, \ell/d).
\]

In the general case (i.e., not in \( F_g \)) it is possible for a necklace and its reversal to be equal, up to rotation – consider the necklace \( ababb \) and its reversal \( bbaba \). However, in \( F_g \) a reduced necklace cannot be equal to its inverse or any of its inverse’s rotations. More generally, we have the following result.

**Lemma 2.2.** Let \( w \) be a freely reduced word of length \( \ell > 0 \) in \( F_g \). Then no conjugate of \( w^{-1} \) equals \( w \).

**Proof.** Let \( w = s_1 \cdots s_\ell \), and write 1 for the empty word and \( \bar{x} \) for \( x^{-1} \).

(i) We first prove that \( w \neq \bar{w} \). If \( w = \bar{w} \) then \( w \) is its own inverse and so \( ww = 1 \). Now put \( w = \bar{w}uv \), where \( u, v \in F_g \) are freely reduced and \( u \) has maximal length. Since \( w \) is freely reduced, \( v \) is non-empty and, by \( u \)'s maximality, there is no free reduction in \( vv \). Thus \( ww = \bar{w}uvu = \bar{w}uv \) is a freely reduced non-empty word, contradicting \( ww = 1 \).

(ii) Now assume that \( w_r = \bar{s}_k \bar{s}_1 \bar{s}_\ell \cdots \bar{s}_k+1, 1 \leq k \leq \ell - 1 \), is a proper rotation of \( \bar{w} \) which equals \( w \). If \( \bar{s}_1 \bar{s}_\ell = 1 \) then free reduction of \( w_r \) yields a word of length \( n < \ell \), so \( w \neq w_r \). Thus \( w_r \) must be freely reduced, and \( w = w_r \) implies that \( w_1 = \bar{s}_k \cdots \bar{s}_1 = s_1 \cdots s_k = \bar{w}_1 \). However this is impossible (part (i), with \( w = w_1 \)), so \( \bar{w} \) is not a proper rotation of \( w \).

(iii) Now consider arbitrary conjugation of \( \bar{w} \), followed by free reduction. If this yields a word of length \( n \neq \ell \), then \( w \neq \bar{w} \). If not, then part (i) or (ii) applies.

Thus the set of reduced necklaces in \( F_g \) of length \( \ell \) can be partitioned into pairs, where the words in a pair are inverses (up to rotation) and are not equal under rotation. So precisely one member of each pair is a bracelet, and the number of reduced bracelets of length \( \ell \) in \( F_g \) is \( CC(g, \ell)/2 \).

In some applications we may only be interested in aperiodic (or prime) words, and it is trivial to modify our algorithms to generate these (see Section 5). From [4, Equation (2.2)] we have the following result.

**Theorem 2.3.** Let \( \mu \) denote the M"obius function. Then the number of reduced prime words of length \( \ell \) in \( F_g \) is equal to

\[
\tau(g, \ell) = \sum_{d|\ell} \mu \left( \frac{\ell}{d} \right) C(g, d).
\]

Thus, in \( F_g \) there are \( \tau(g, \ell)/\ell \) reduced necklaces and \( \tau(g, \ell)/2\ell \) reduced bracelets of length \( \ell \) which are not proper powers.

For \( F_1 \), with generator \( z \), there are only two reduced necklaces \( (z^\ell) \) and \( (z^{-1})^\ell \) and one reduced bracelet \( (z^\ell) \) for all \( \ell > 0 \), so we ignore this case and assume throughout
Algorithm 1 The areInv() function
1: function areInv( x, y : integer ) returns boolean;
2: begin
3:   if x mod 2 = 0 then
4:     begin if x + 1 = y then return TRUE; end;
5:   else
6:     begin if x = y + 1 then return TRUE; end;
7:   return FALSE;
8: end;

Algorithm 2 The getInv() function
1: function getInv( x : integer ) returns integer;
2: begin
3:   if x mod 2 = 0 then return x + 1;
4:   return x − 1;
5: end;

that \( g > 1 \). Obviously, for \( g = 1 \) and \( \ell > 1 \) there are no reduced prime words. The algorithms we give are valid for \( g = 1 \) but they are not CAT, since it takes \( O(\ell) \) time to set the word to \( z \cdots z \) or to \( z^{-1} \cdots z^{-1} \).

3. Listing Necklaces. We first need to decide on the conventions we adopt for representing the group generators and their inverses, and how to order these \( k \) symbols. Using the integers \( \pm 1, \ldots, \pm g \) is straightforward and is convenient for generating and checking inverses. However, it is awkward for running through the symbols in order and checking symbol ordering, since we require a generator to precede its inverse and for \( \pm 1 \) to precede +2, etc. Accordingly, we use the integers 0, \( \ldots, k − 1 \) with the usual numeric ordering, where even integers \( j \) denote the generators and odd integers \( j + 1 \) their inverses.

To handle inverses we introduce the two utility functions areInv() and getInv() of Algorithms 1 and 2. These, respectively, check whether or not two symbols are inverses and return the inverse of a symbol. These functions are common to both the necklace and bracelet algorithms and run in constant time. We separate out these functions for simplicity — in practice they can be compiled as inline functions or replaced by macros.

Our recursive necklace generation procedure genNeck() in Algorithm 3 is now a simple modification of [3, Algorithm 2.1], with the if statements at Lines 9 and 13 ensuring that the prenecklaces remain freely reduced and that the final necklaces are cyclically reduced. For clarity we do not use the guard value \( a_0 \) of [3] and instead use the wrapper code of Algorithm 4. This explicitly sets \( a_1 \) (recording its inverse in aoi to facilitate cyclic reduction checking) and ensures that the prenecklaces at entry to genNeck() are non-empty.

4. Listing Bracelets. Our bracelet algorithm is inspired by that in [13], with reversal of a prenecklace being replaced by word inversion (i.e., reversal and element-to-inverse mapping). The recursive bracelet generation procedure genBrace() and its wrapper code are given in Algorithms 5 and 6, while Algorithm 7 is the checkInv() function for comparing a prenecklace with its inverse. The genBrace() procedure is an augmented version of genNeck(), with the additional code checking each prenecklace...
Algorithm 3 The genNeck() procedure

1: procedure genNeck( t, p : integer );
2: local j : integer;
3: begin
4: if t > ℓ then begin
5: if ℓ mod p = 0 then process necklace a₁ ··· aₖ;
6: return;
7: end;
8: j := aₜ₋₋₁;
9: if not areInv( aₜ₋₁, j ) and ( t < ℓ or j ≠ aoi ) then begin
10: aₜ := j; genNeck( t + 1, p );
11: end;
12: for j from aₜ₋₋₁ + 1 to k − 1 do begin
13: if not areInv( aₜ₋₁, j ) and ( t < ℓ or j ≠ aoi ) then begin
14: aₜ := j; genNeck( t + 1, t );
15: end;
16: end;
17: return;
18: end;

Algorithm 4 The necklace code wrapper

1: global ℓ, g, k, aoi : integer; a : integer array;
2: local j : integer;
3: g := number of generators; ℓ := word length;
4: k := 2g;
5: for j from 0 to k − 1 do begin
6: a₁ := j;
7: aoi := getInv( j );
8: genNeck( 2,1 );
9: end;

(i.e., the putative “prebracelets”) against its inverse and rejecting those that cannot
yield bracelets. Thus each rotation of the inverse of the final words is tested, and
necklaces which are not bracelets are not generated.

The use of inverses as opposed to reversals actually results in a somewhat simpler
algorithm compared with that in [13]. Firstly, note that our order is chosen so that
a generator immediately precedes its inverse. This implies that bracelets cannot
start with a generator inverse and so these can be skipped in the wrapper code.
Secondly, although the checkInv() function can return “equal” as well as “less than”
or “greater than” (as does the CheckRev() function in [13]), by Lemma 2.2 it never
does so since our preneckaces are always freely reduced. This simplifies the code in
the genBrace() procedure, which needs only four parameters compared with the six
of GenBracelets() in [13].

The t and p arguments of genBrace() are, respectively, the index of the next
position in the array a and the length of the longest prefix of a that is a Lyndon
word. The u and v arguments are, respectively, the number of copies of a₁ at the
start of a and the number of copies of a₁⁻¹ = aoi at the end of a, with the initial value
of v saved in the local variable vv. The code at Lines 9 to 14 and Lines 22 and 25
Algorithm 5 The genBrace() procedure

1: procedure genBrace( t, p, u, v : integer );
2: local j, vv := v : integer;
3: begin
4:   if t > ℓ then begin
5:     if ℓ mod p = 0 then process bracelet a₁ ··· aₖ;
6:     return;
7:   end;
8:   j := aₜ−p;
9:   if j = a₁ then
10:     begin v := 0; if u = t − 1 then u := u + 1; end;
11:   else if j = aoi then
12:     v := v + 1;
13:   else
14:     v := 0;
15:   if not areInv( aₜ−1, j ) and ( t < ℓ or j ≠ aoi ) then begin
16:     aₜ := j;
17:     if u = v then
18:       begin if checkInv( t, u + 1 ) < 0 then genBrace( t + 1, p, u, v ); end;
19:     else if u > v then
20:       genBrace( t + 1, p, u, v );
21:     end;
22:   if u = t then u := u − 1;
23:   for j from aₜ−p + 1 to k − 1 do begin
24:     if not areInv( aₜ−1, j ) and ( t < ℓ or j ≠ aoi ) then begin
25:     if j = aoi then v := vv + 1; else v := 0;
26:     aₜ := j;
27:     if u = v then
28:       begin if checkInv( t, u + 1 ) < 0 then genBrace( t + 1, t, u, v ); end;
29:     else if u > v then
30:       genBrace( t + 1, t, u, v );
31:     end;
32:   end;
33:   return;
34: end;

adjusts u and v as necessary for the next call to genBrace() (if any), using the current value of j (i.e., the next potential value for aₜ) and the current word length t − 1.

The recursive calls to genBrace() are inside the if-statements of Lines 17 to 20 and Lines 27 to 30. If u > v then the current prenecklace is less than its inverse, so we can immediately call genBrace(). If u < v then the inverse is less than the prenecklace, the prenecklace cannot yield a bracelet, so we do nothing. If u = v then we need to call the checkInv() function to compare the prenecklace with its inverse. If the prenecklace is less we call genBrace(), otherwise we do nothing.

When the checkInv() function is called, the current prenecklace starts with u copies of a₁ and ends with v = u copies of aoi. The remainder, γ, is non-empty, does not start with a₁ or end with aoi, and is freely reduced. The arguments t and i are the current prenecklace length and the index of the start of γ. The for-loop compares
Algorithm 6 The bracelet code wrapper
1: global $\ell, g, k, aoi$ : integer; $a$ : integer array;
2: local $j$ : integer;
3: $g :=$ number of generators; $\ell :=$ word length;
4: $k := 2g$;
5: for $j$ from 0 to $k - 2$ step 2 do begin
6:    $a_1 := j$;
7:    $aoi := \text{getInv}( j )$;
8:    genBrace( 2,1,1,0 );
9: end;

Algorithm 7 The checkInv() function
1: function checkInv( $t, i$ : integer ) returns integer;
2: local $j$ : integer;
3: begin
4:    for $j$ from $i$ to $t$ do begin
5:        if $a_j < \text{getInv}( a_{t-j+1} )$ then return $-1$;
6:        if $a_j > \text{getInv}( a_{t-j+1} )$ then return $+1$;
7:    end;
8:    return 0;
9: end;

γ with its inverse, returning $-1$ if $\gamma$ (and thus the prenecklace) precedes its inverse and $+1$ if $\gamma^{-1}$ precedes $\gamma$. The upper limit on the for-loop is simply a convenient placeholder – the loop is guaranteed to return $-1$ or $+1$ for some $i \leq \lfloor (t + 1)/2 \rfloor$.

5. Concluding Remarks. We have implemented our algorithms in the C and Magma languages and incorporated them into programs for generating and testing lists of conjugacy classes and presentations. The generation of word lists has proved very fast, with the programs' running times being dominated by the times to process the necklaces and bracelets in the lists. Empirical evidence (see Appendix A) suggests that our algorithms are CAT, but we have no proof of this.

The genNeck() and genBrace() procedures as given process all reduced necklaces and bracelets. If the "$\ell \mod p = 0$" tests are replaced by "$\ell = p$" then only the reduced aperiodic necklaces (Lyndon words) and bracelets are processed by these procedures.

The recursive nature of our necklace and bracelet algorithms induces a tree structure on their search spaces. These trees can easily be split into subtrees, allowing an enumeration to be parallelised [3] or distributed across a set of heterogeneous machines.

Appendix A. Complexity Tests. Implementations of our necklace and bracelet listing algorithms for reduced words in groups have proved very effective. However we have no proof that they run in constant amortized time. Accordingly, in a similar manner to [14, §5 & §6.2.5], we produced experimental results for the amount of work done compared with the number of necklaces or bracelets generated. For these tests the "process . . . $a_1 \cdots a_\ell$" actions in genNeck() and genBrace() were replaced by code to accumulate the total number of necklaces and of bracelets. These counts were checked against the expected counts from Section 2 and used to calculate the work per necklace and per bracelet data.
The areInv() and getInv() functions are used by both algorithms and are constant time. Each algorithm starts with a for-loop, each iteration of which makes a call to genNeck() or to genBrace(). For necklaces we count the total number of calls, both direct and recursive, to genNeck(). Apart from its embedded for-loop, each call to genNeck() is constant time. So our measure of the amount of work done is the total number of calls to genNeck() plus the total number of iterations of the embedded for-loop across all calls to genNeck(). Figure 1 plots, for various values
of the number of group generators $g$, the word length against the ratio of total work to number of necklaces.

For bracelets we count the total number of calls to `genBrace()` and the total number of iterations of its embedded `for`-loop. We also need to account for the `checkInv()` function, which is not constant time. This function is only called if $u$ and $v$ (in `genBrace()`) are equal, so at least one iteration of `checkInv()`’s embedded `for`-loop is guaranteed for each call. Thus, we count the total number of iterations of this loop across all calls to `checkInv()` and add this to our total. Figure 2 plots, for various values of the number of group generators $g$, the word length against the ratio of total work to number of bracelets.

For both necklaces and bracelets, for all $2 \leq g \leq 6$, the ratio of the total amount of work done to the number of reduced words generated is decreasing (after an initial peak) as the word length increases. This strongly suggests that the algorithms are CAT.

REFERENCES

[1] GAP – Groups, Algorithms, and Programming, Version 4.8.7, 2017, http://www.gap-system.org.
[2] W. Bosma, J. Cannon, and C. Playoust, The Magma algebra system, I: The user language, J. Symbolic Comput., 24 (1997), pp. 235–265.
[3] K. Cattell, F. Ruskey, J. Sawada, M. Serra, and C. R. Miers, Fast algorithms to generate necklaces, unlabeled necklaces, and irreducible polynomials over GF(2), J. Algorithms, 37 (2000), pp. 267–282.
[4] M. Coornaert, Asymptotic growth of conjugacy classes in finitely-generated free groups, Internat. J. Algebra Comput., 15 (2005), pp. 887–892.
[5] J. P. Duval, Factorizing words over an ordered alphabet, J. Algorithms, 4 (1983), pp. 363–381.
[6] H. Fredericksen and I. J. Kessler, An algorithm for generating necklaces of beads in two colors, Discrete Math., 61 (1986), pp. 181–188.
[7] H. Fredericksen and J. Maiorana, Necklaces of beads in $k$ colors and $k$-ary de Bruijn sequences, Discrete Math., 23 (1978), pp. 207–210.
[8] R. C. Lyndon and P. E. Schupp, Combinatorial Group Theory, Springer-Verlag, 1977.
[9] W. Magnus, A. Karrass, and D. Solitar, Combinatorial Group Theory, John Wiley & Sons, 1966.
[10] I. Rivin, Growth in free groups (and other stories), 1999, https://arxiv.org/abs/math/9911076.
[11] F. Ruskey, C. Savage, and T. M. Wang, Generating necklaces, J. Algorithms, 13 (1992), pp. 414–430.
[12] F. Ruskey and J. Sawada, Generating necklaces and strings with forbidden substrings, in Computing and Combinatorics, D.-Z. Du, P. Eades, V. Estivill-Castro, X. Lin, and A. Sharma, eds., vol. 1858 of Lecture Notes in Comput. Sci., Springer-Verlag, New York, 2000, pp. 330–339.
[13] J. Sawada, Generating bracelets in constant amortized time, SIAM J. Comput., 31 (2001), pp. 259–268.
[14] J. Sawada, A fast algorithm for generating nonisomorphic chord diagrams, SIAM J. Discrete Math., 15 (2002), pp. 546–561.