

Combinatorics on Words

Suppression of Unfavorable Factors in Pattern Avoidance

Veikko Keränen

We explain extensive computer-aided searches that have been carried out for many years to find new ways of constructing abelian square-free words over four letters. These structures have turned out to be very rare and hard to find. We also encountered highly nonlinear phenomena that considerably affected our calculations and usually made them hard to accomplish. However, quite recently, we gained new insight into why these structures are so very rare. Consequently, the present work has the potential to make future explorations easier. The rarity of long words that avoid abelian squares can be explained, at least partly, by using the concept of an unfavorable factor. The purpose of this article is to describe the use of *Mathematica* in searching for and suppressing these factors. In principle, the same programs can be used with slight modifications for other kinds of word patterns as well.

■ Introduction

An abelian square is a nonempty word uv , where u and v are permutations (anagrams) of each other. In 1961, Paul Erdős [1] raised the question of whether or not abelian squares can be avoided (as factors) in infinitely long words (also called strings). In 1969, Pleasants [2] solved positively the question by Erdős in the case of a five-letter alphabet, but the four-letter case remained open until 1992 when the author [3] presented an abelian square-free (a-2-free) endomorphism g_{85} over the four-letter alphabet $\Sigma_4 = \{a, b, c, d\}$. This endomorphism g_{85} was found after long computer experiments, and, for a long time, all known methods for constructing arbitrarily long a-2-free words on Σ_4 were based on the structure of g_{85} , including Carpi's [4] modification from 1998.

In 2002, after over 11 years of exhaustive searches, we found a completely new endomorphism g_{98} of Σ_4^* , the iteration of which produces an infinite a-2-free word. The endomorphism g_{98} is not an a-2-free endomorphism itself, since it does not preserve the a-2-freeness of all words of length 7. However, g_{98} can be used with g_{85} to produce a-2-free DTOL-languages of unlimited size. For the latest remarkable development, the reader is referred to the author's publications [5, 6, 7]. Nowadays, a-2-free words have also found their way to applications, for instance, in number theory, algorithmic music, and only recently in cryptography (Rivest [8] in 2005, Andreeva et al. [9] in 2008).

An important analogous open avoidability problem for the three-letter case was posed by Sami Mäkelä [10] in 2002. In this problem, shortest possible abelian

squares, that is, repetitions of the form xx or xxx for a letter $x \in \Sigma_3 = \{a, b, c\}$, are allowed to occur.

Quite recently we have gained new insight into why these structures are so very rare. We are now able to explain, at least partly, the rarity of long words that avoid abelian squares by using the concept of an unfavorable factor. The purpose of this article is to describe the use of *Mathematica* in searching for and suppressing these factors. In principle, the same programs can be used with slight modifications for other kinds of (possibly nonabelian) word patterns as well.

The concept of an unfavorable factor, together with our programs, can be used in connection with efficient matrix methods presented by Ochem and Reix [11] to find upper bounds for the growth of the number of repetition-free words. Indeed, their approach is also directly applicable to the abelian square-freeness case. The related computations are still proceeding, and, consequently, this topic is not elaborated here.

We have carefully studied the options to make the programs efficient with regard to running time and the required memory space. At the same time, for further development, the structures need to be flexible. Indeed, it is expected that these programs will be run interactively for quite a long time. The programs are also likely to be transformed into various computational environments (such as GRID, C++, FPGA). This work has already been started. In the process of these computations, efficiency will increase due to the natural reduction process. Indeed, the long lists of words that we now have to use can most likely be considerably reduced as one moves on to study longer words.

The programs and some of the structures involved are quite complicated and would have been very hard to develop without using *Mathematica*. We have been using integer coding and cumulative integer lists for words incorporated into quite extensive precomputations. Moreover, certain symbolic representations for words and *Mathematica*'s pattern matching properties (for lists and strings) have been extremely useful.

At the same time, we feel that it would be extremely beneficial for programmability and computational efficiency if in *Mathematica* there were an efficient way of saying, for example, that when matching the pattern $\{___, u__, v__\}$, we are actually interested only in those cases of u and v in which their lengths are equal. Indeed, our words can contain thousands of letters, and the absence of restricted pattern matching led us to write part of the programs in a quite tedious C-like fashion. Fortunately, to our big relief, debugging turned out to be a comfortable process. In the future, the new integrated string functions of *Mathematica* like `RegularExpression` can also be used.

We define an *unfavorable* (or *forbidden*) word or factor to be an a -2-free word over a fixed alphabet Σ (in our case $\Sigma = \Sigma_4 = \{a, b, c, d\}$, or $\Sigma = \Sigma_3 = \{a, b, c\}$, in which case we allow xx or xxx for a letter x) if it cannot occur as a proper factor inside any infinite a -2-free word. That is to say, over Σ , an unfavorable a -2-free word cannot be continued to an infinitely long word to the left and to the right without necessarily creating an abelian square at some point. (However, it might well be possible to extend such a word boundlessly in one direction, say to the right, without producing any abelian squares. Experiments support this conjecture, but the existence of such unfavorable factors remains an open question.)

We now give a short explanation of our search for unfavorable factors. Let the alphabet Σ be fixed and consider words over it. We take a word (we actually need to consider all the a-2-free words of a given length) and try to extend it in an a-2-free fashion to the right and left in all possible ways up to a given upper bound for the total length. Each time, the length of the word increases only by a given fixed length. We extend alternately to the right and left, and backtrack when necessary. If the upper bounds are reached, then the original word is *so-far-favorable* (it may still turn out to be unfavorable after more experiments). If there is no way to reach the upper bounds, then the original word is classified, without any doubt, as unfavorable. Thus, for a given length, we obtain three kinds of words: *unfavorable* (*bad*), *so-far-favorable* (*so-far-so-good*), and *favorable* (*good*). The latter type contains words occurring as factors in a-2-free words obtained by using, for example, g_{85} , g_{98} , and Carpi's modification of g_{85} .

It is a remarkable phenomenon that relatively short so-far-favorable words turn out to be unfavorable factors after being "safely" extendable (to the right and left) for a long distance (and with a huge number of branches). One might have expected the long buffers to be sufficient for further growth. Due to Carpi [4] and Keränen [7], we know that the number of a-2-free words over four letters grows exponentially (greater than or equal to 1.02306^n) with respect to word length n . But how do these words grow? We conjecture that in spite of the exponential growth, the ratio between the number of properly extendable, that is, favorable, words of length n , and the number of unfavorable factors of the same length, tends to zero as the length n tends to infinity! Thus, we suspect that the vast majority of a-2-free words over four (and, arguably, three) letters cannot occur as proper factors in the middle of very long (infinite) words. In a way, this would also explain why it is so extremely difficult to find a-2-free endomorphisms over four letters. At present we know, for example, that just a little over half (50.5737%) of the a-2-free words over Σ_4 of length 20 are indeed unfavorable. In the future, this and other similar observations could lead to a better understanding of the abelian square-free structures.

As an example, in the case of four letters, the following words together with their permutations and mirror images are unfavorable (forbidden) factors.

```
In[152]:= unfavourableFactorsOfLength8 =
{"abacdaba", "abacdbab", "abcabdab", "abcabdcb", "abcadbad"};

In[153]:= someUnfavourableFactorsOfLength20 =
{"abacabadbacbcdbacdbd", "abacabdadcdbcacbcd", "abacadcabcbdcabdcda
d", "abacdabcbdbcadabdacd",
"abcabadbcbdbcbcabcd", "abcabadbcbdbcbcabcd", "abcacadbcdadbabcabda
", "abcacadbcdadbabcabda",
"abcacdbacdadbabcabda", "abcacdbcbdbcdacdcbad", "abcabcbdcacdbdababc
", "abcabdbacacadbcdadb",
"abcbadbcacdcabdbcdba", "abcbadcbdbcdacabcadcb", "abcdbcbcbcdcdacbdac
", "abcdbdadcbcadabdadb"};
```

These words form a very interesting case, leading to a million-fold fluctuational phenomenon (with respect to the required running time and memory space, if we wish to see the tree of all possibilities). This, actually quite frequently appearing, case is illustrated (in a somewhat different setting than the one we use elsewhere in this article) online at [12]. An example is the following illustration for

"abcdbdbcbacbdcdacbdac". Note the top at (82, 84218) and the death of all branches at word length 87.

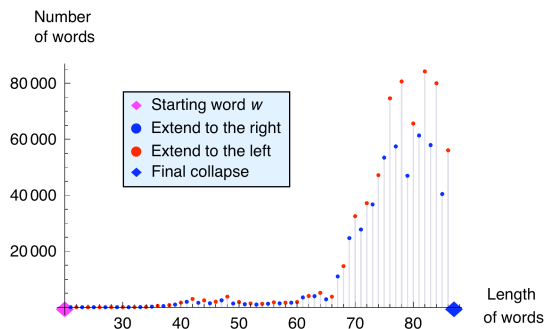


Figure 1. Extend `abcdbdbcbacbdcdacbdac` alternately to the right and left, stepping one letter at a time.

We now provide an explanation for the case shown in the preceding figure. We tried to extend the word $w = \text{"abcdbdbcbacbdcdacbdac"}$ to the right (blue) and to the left (red) with all possible letters over Σ_4 . The length of the words increases only by one letter at a time (alternately to the right and left). All possible words are gathered in a list and the number of words is plotted according to their length. For each word there are four possibilities: it can be continued with 3, 2, 1, or 0 letters. The case of 0 letters means that the word cannot be continued (in an a -2-free way) at all. If no words can be continued, then we get an empty list and the computation ends. The resulting list consists merely of unfavorable (or bad, or forbidden) factors that start from the original given word, and this starting word is “in the middle” of every nonempty word in the list. Indeed, all the computations for the words in the list of `someUnfavourableFactors`: `OfLength20`, together with a great many other similar words, do end with an empty list (i.e., with a dead end) even though the width of the bidirectional tree, and the computational time, can be huge. All this can make finding unfavorable factors really challenging.

Here are some additional illustrations of this phenomenon. This time, the number of words is depicted using logarithmic scaling.

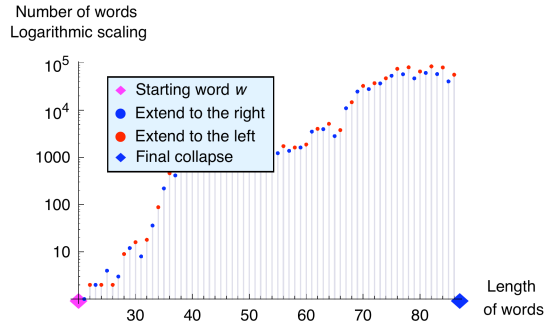


Figure 2. Extend abacadbcabdcadcbab alternately to the right and left, stepping one letter at a time.

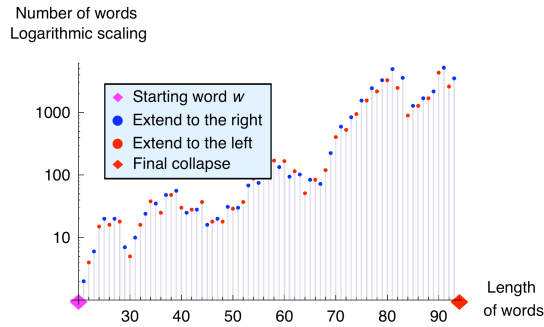


Figure 3. Extend abacadbdadcacbadcbab alternately to the right and left, stepping one letter at a time.

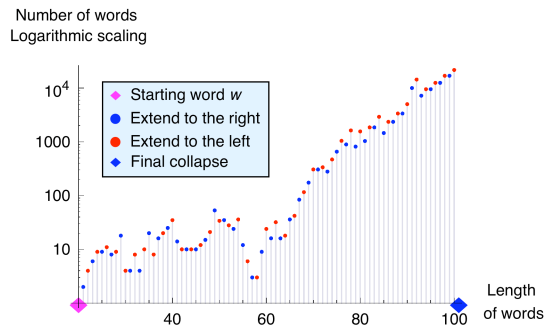


Figure 4. Extend abacdbacbcdbcadacdbab alternately to the right and left, stepping one letter at a time.

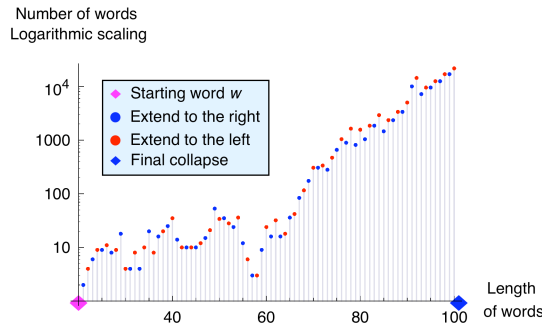


Figure 5. Extend $abcdadbdbdcdbacadbacab$ alternately to the right and left, stepping one letter at a time.

In the last illustration, the word $w = "abcdadbdbdbacadbdbcbdad"$ of length 22 turns out to be unfavorable in a monstrous way. The blue point, just before the final collapse, is at $(117, 7866918)$. In spite of this, not a single extension is possible to the left (which would otherwise lead to words of length 118).

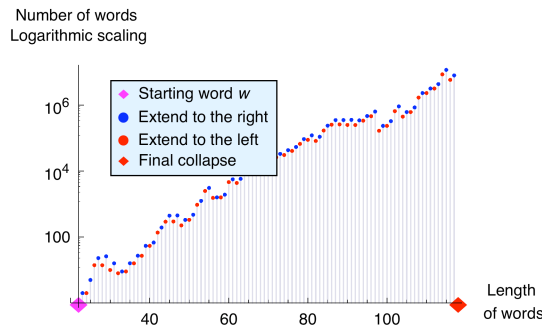


Figure 6. Extend $abcdadbdbdbacadbcbdad$ alternately to the right and left, stepping one letter at a time.

In passing, we mention that some unfavorable factors can even be cut shorter from the right or left ends to (shortened) factors which, after a transitional phase, have the same final trajectory.

■ Preliminaries

In this section we present some mathematical notations and terminology. Our terminology is more or less standard in the field of combinatorics on words. Consequently, the reader might consult this section later, if need arises.

An *alphabet* Σ is a finite nonempty set of abstract symbols called *letters*. A *word* (*string*) over Σ is a finite (unless otherwise indicated) string, or sequence, of letters belonging to Σ . The set of all words (nonempty words) over Σ is denoted by Σ^* (Σ^+). On the set Σ^* , the associative binary operation of *catenation* is defined.

For words u and v , it is the juxtaposition uv . The *empty word*, which is the neutral element of catenation, is denoted by λ . The algebraic structures Σ^* and Σ^+ are called, respectively, the free monoid and the free semigroup generated by Σ .

Let $w = x_1 \dots x_m$, $x_i \in \Sigma$. The *length* of the word w , denoted by $|w|$, is the number of occurrences of letters in w , that is, $|w| = m$. Let $\Sigma = \{a_1, \dots, a_n\}$. The number of occurrences of one letter $x \in \Sigma$ in w is denoted by $|w|_x$, or simply by $|w|_i$ if $x = a_i$. The notation $\psi_\Sigma(w)$ stands for the *Parikh vector* of w , that is, $\psi_\Sigma(w) = (|w|_1, \dots, |w|_n)$. Usually we will omit the subscript Σ and write ψ instead of ψ_Σ . Quite interchangeably with the Parikh vector notation, we also use formal sums $\psi_s(w) = \sum_{x \in \{a_1, \dots, a_n\}} k_x x$, with $k_x \in \mathbb{N}$. For example, $\psi_s(abacaba) = 4a + 2b + c$. Thus $\psi_s(w)$, with w a word over Σ , is an element of the abelian free monoid \mathbb{N}^Σ generated by Σ . We will also consider differences of Parikh vectors and differences of formal sums. Consequently, these vectors and sums are extended into elements of the abelian free group \mathbb{Z}^Σ generated by Σ . The neutral element of \mathbb{Z}^Σ is denoted by $\mathbf{0}$.

A word u is called a *factor* (some authors call it a *subword*) of a word w , if $w = p u s$ for some words p and s . If $p = \lambda$ ($s = \lambda$), then u is called a *prefix* (a *suffix*) of w .

Let $k \geq 2$ be a given integer. A *k-repetition* (an *abelian k-repetition*) is a nonempty word of the form $R^k (P_1 \dots P_k)$, where $\psi(P_\mu) = \psi(P_\nu)$ for all $1 \leq \mu < \nu \leq k$, that is, the P_i are permutations of each other). Instead of (abelian) 2- and 3-repetitions, the terms (*abelian*) *squares* and (*abelian*) *cubes* are often used. A word or an ω -word (explained later) is called *k-repetition free* (*abelian k-repetition free*, or *k-free in the abelian sense*), or in short *k-free* (*a-k-free*), if it does not contain any *k-repetition* (abelian *k-repetition*) as a factor. A word sequence or a word set is *k-free* (*a-k-free*) if all words in it are *k-free* (*a-k-free*). If, for a fixed k , it is possible to construct arbitrarily long (infinite) *a-k-free* (or other pattern-free) words over a given alphabet Σ , then we say that abelian *k-repetitions* (or those patterns) are *avoidable* over Σ .

A *morphism* h is a mapping between free monoids Σ^* and Δ^* with $h(uv) = h(u)h(v)$ for every u and v in Σ^* . In particular, $h(\lambda) = \lambda$. A morphism $h: \Sigma^* \rightarrow \Delta^*$, being compatible with the catenation of words, is uniquely defined if the word $h(x) \in \Delta^*$ is (effectively) given for each $x \in \Sigma$. If $\Delta = \Sigma$, we call h an *endomorphism* (and usually write g instead of h). For a morphism h and a language L , we define $h(L) = \{h(w) \mid w \in L\}$. A morphism h is termed *uniformly growing* if $|h(x)| = |h(y)| \geq 2$ for every x and $y \in \Sigma$.

For a given integer $k \geq 2$, a morphism $h: \Sigma^* \rightarrow \Delta^*$ is called *k-free* (*a-k-free*) if $h(w)$ is *k-free* (*a-k-free*) for every *k-free* (*a-k-free*) word $w \in \Sigma^*$.

With regards to *L-systems* (Aristid Lindenmayer 1925-1989), we specify the following concepts. A *DOL-system* is a triple $G = (\Sigma, g, \alpha_0)$, where Σ is an alphabet, $g: \Sigma^* \rightarrow \Sigma^*$ is an endomorphism, and α_0 , called the *axiom*, is a word over Σ . The (*word*) *sequence* $S(G)$ generated by G consists of the words

$$\alpha_0 = g^0(\alpha_0), g^1(\alpha_0), g^2(\alpha_0), g^3(\alpha_0), \dots,$$

where $g^i(\alpha_0) = g(g^{i-1}(\alpha_0))$ for $i \geq 1$. The *language* of G is defined by $L(G) = \{g^i(\alpha_0) \mid i \geq 0\}$. Languages (sequences) defined by a DOL-system are re-

ferred to as *D0L-languages* (*D0L-sequences*). D0L-systems provide a very convenient way for defining languages and infinite words. Furthermore, if g and α_0 are k -free (a- k -free), then the iteration of g will yield a k -free (a- k -free) D0L-sequence. An *HD0L-system* is a 5-tuple $G_1 = (\Sigma, \Delta, g, b, \alpha_0)$, where (Σ, g, α_0) is a D0L-system, called the underlying D0L-system of G_1 , Δ is an alphabet, and $b: \Sigma^* \rightarrow \Delta^*$ is a morphism. The *HD0L-sequence* $S(G_1)$ generated by G_1 consists of the words

$$b(\alpha_0) = b(g^0(\alpha_0)), b(g^1(\alpha_0)), b(g^2(\alpha_0)), b(g^3(\alpha_0)), \dots,$$

and the *HD0L-language* of G_1 is the set $L(G_1) = \{b(g^i(\alpha_0)) \mid i \geq 0\}$. A *DT0L-system* is a triple $G_2 = (\Sigma, H, \alpha_0)$, where H is a finite nonempty set of morphisms (called tables) and (Σ, b, α_0) is a D0L-system for every $b \in H$. The *DT0L-language* of G_2 is the set $L(G_2) = \{w \mid w = \alpha_0 \text{ or } w = b_k \dots b_1(\alpha_0), \text{ where the compositions } b_k \dots b_1 \text{ of morphisms are constructed from } b_1, \dots, b_k \in H\}$. Obviously, a DT0L-system can be regarded as a D0L-system, when H contains only one (endo)morphism. For a thorough discussion of various L-systems the reader is referred to Rozenberg and Salomaa [13].

An ω -word is an infinite sequence, from left to right, of letters of an alphabet Σ . Thus an ω -word can be identified with a mapping of \mathbb{N}_+ into Σ . One can construct an ω -word, for example, by iterating an endomorphism $g: \Sigma^* \rightarrow \Sigma^*$, such that $\lambda \notin g(\Sigma)$ and $g(x) = xw$ for some $x \in \Sigma, w \in \Sigma^+$. Such a morphism g is called *prefix preserving* for the reason that $g^i(x)$ is a proper prefix of $g^{i+1}(x)$ whenever $i \geq 0$. An ω -word is obtained as the "limit" of the sequence $g^i(a); i = 0, 1, 2, \dots$.

■ Suppression of Unfavorable Factors with *Mathematica*

In this section we give examples of the developed *Mathematica* program. The correctness of the program and related packages has been tested, but exhaustive computer runs are still likely to take a long time in the future. The full program and further versions of it can be downloaded from [14] or from [15], the latter of which is a general link page for the topic. In addition, the program (in original form) is included at the end of this article, and the reader is invited to experiment with it. For this purpose, one may copy and edit the Input cells of the examples presented. The program consists of initialization cells and will be activated automatically.

In the following examples, many of the function definitions are omitted (they can be found at the end of this article). Note that all of the structures, variables, and constants starting with ϵ are global. For example, the global variable ϵstate represents the state of the construction. Its values are strings, including, for example, "extendOrChangeRight", "extendOrChangeLeft", "testRight", "testLeft", "failBoth", and "succeeded".

As mentioned in the Introduction, we first fix the alphabet Σ and consider words over it. We take a word (in the final investigation, we will actually take all of the a-2-free words of a given length) and try to extend it in an a-2-free fashion to the right and left in all possible ways up to a given upper bound for the total length.

Each time, the length of the word increases only by a given fixed length. We extend alternately to the right and left, and backtrack when necessary. If the upper bounds are reached, then the original word is so-far-favorable. If there is no way to reach the upper bounds, then the original word is classified, without any doubt, to be unfavorable. At present, favorable words (for the four-letter case) consist of only those occurring as factors in a -2-free words obtained by using known a -2-free endomorphisms and substitutions such as g_{85} , g_{98} , Carpi's [4] modification of g_{85} , and the new examples presented in [7].

In the final program we use integer coding for letters of the alphabet Σ and cumulative integer lists for words. This makes detecting abelian squares fast. We use two different cumulative integer lists, `€cumulIntListRight` for the right-hand extensions and `€cumulIntListLeft` for the left-hand extensions. This makes the addition of integers (addition of cumulative integer lists, in fact) fast, but forces us to write the testing function `testA2RLpairCumulIntList` in a more complicated fashion (nevertheless, still maintaining the necessary high speed). In building all the structures, quite extensive precomputations are needed.

Now we define the alphabets by using letters (strings of length 1) and integers.

```
In[154]:= €alphTwoLet = {"a", "b"};
          €alphTwoInt = {0, 1};
          €alphThreeLet = {"a", "b", "c"};
          €alphThreeInt = {0, 1, 216};
          (* Words of length ≤ (216-1)*4/3 = 87380
             are safe to use - provided that they do
             not contain xxxx for a letter x in €alphThreeLet *)
          €alphFourLet = {"a", "b", "c", "d"};
          €alphFourInt = {0, 1, 210, 220};
          (* Words of length ≤ (210-1)*2 = 2046
             are safe to use - provided that they do
             not contain xx for a letter x in €alphFourLet *)
          €alphThreeLetInt = {€alphTwoLet, €alphTwoInt};
          €alphThreeLetInt = {€alphThreeLet, €alphThreeInt};
          €alphFourLetInt = {€alphFourLet, €alphFourInt};
```

This first example using cumulative Parikh vectors is symbolic (Parikh vectors are explained in the Preliminaries section).

```
In[163]:= convertStrToCumulIntList["bacabcacab",
          {"a", "b", "c"}, {"a", "b", "c"}]
Out[163]:= {b, a + b, a + b + c, 2 a + b + c, 2 a + 2 b + c, 2 a + 2 b + 2 c,
          3 a + 2 b + 2 c, 3 a + 2 b + 3 c, 4 a + 2 b + 3 c, 4 a + 3 b + 3 c}
```

In actual computations we use the integer representation for these cumulative Parikh vectors.

```
In[164]:= convertStrToCumulIntList["bacabcacab", €alphThreeLetInt]
Out[164]:= {1, 1, 65 537, 65 537, 65 538, 131 074, 131 074, 196 610, 196 610, 196 611}
```

Then we transform back to the string representation over three letters.

```
In[165]:= convertCumulIntListToStr[%, €alphThreeLetInt]
```

```
Out[165]= bacabcacab
```

The case of four letters can be handled in a similar way.

```
In[166]:= convertStrToCumulIntList["bacabdacad", €alphFourLetInt]
```

```
Out[166]= {1, 1, 1025, 1025, 1026, 1 049 602,
           1 049 602, 1 050 626, 1 050 626, 2 099 202}
```

```
In[167]:= convertCumulIntListToStr[%, €alphFourLetInt]
```

```
Out[167]= bacabdacad
```

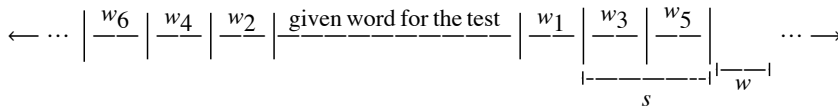


Figure 7. Extend the given word alternately to the right and left. The word sw is favorable or so-far-favorable.

We have also taken care that the selection of the next proper candidate is done in an efficient way. If all the possible candidates for w have already been tried out, then the program backtracks to change the other end's suffix. Indeed, the process is the same for the right- and left-hand extensions (both of their cumulative list representations grow from left to right), so we can really speak of suffixes only. The lengths for the suffix s and the extension w need to be fixed at the beginning of the computation, and changing (re-fixing) the lengths usually requires new quite extensive precomputations.

In the program at the end of this article, the lengths have been set as $|s| = 4$ and $|w| = 4$, but they can be, and usually are, selected differently as well. Up to now, we have been using, for example, the lengths $|s| = 8$, $|w| = 4$, and $|s| = 12$, $|w| = 4$. Longer suffixes s would improve the computational time efficiency considerably, but, on the other hand, the structures might need too much memory in our present environment for distributed computing. Moreover, selecting $|w| = 1$ would allow the maximal avoidance of unfavorable factors in the extensions. However, the setting $|w| > 1$ probably allows detecting the abelian squares more quickly, albeit the final decision for this still requires quite extensive experiments.

In the following example, we add all possible words w of length 4, represented by lists that follow all possible reduced suffixes s of length 4. Here the term “reduced” means that we pay attention only to the structure of the word—the other possibilities can straightforwardly be obtained by permutations, that is, by renaming letters. The example originates from the three-letter case in which short abelian repetitions of the form xx or xxx for a letter $x \in \Sigma_3 = \{a, b, c\}$ are allowed. For simplicity, we show the words in their string form.

The next input serves only descriptive purposes and is not intended for actual evaluation inside this notebook.

```

€reducedWordListSuff4Add4 =
  selectWordsWithProperPrefixes[#, addWordList8StartWitha] & /@
  reducedWordList4

```

To save memory and to make the computations as fast as possible, the final calculations in fact use symbols instead of strings. The cumulative integer lists are associated to symbols by using rules. Moreover, to save memory, the permutations are added only when needed. In the following example, we show a rule from symbols to cumulative integer lists (in an abbreviated form).

```

€ruleSymbolsToCumulIntegerLists = {aaab -> {0, 0, 0, 1}, aaba -> {0, 0, 1, 1},
aabb -> {0, 0, 1, 2}, aabc -> {0, 0, 1, 65537}, abaa -> {0, 1, 1, 1}, abac -> {0,
1, 1, 65537}, abbb -> {0, 1, 2, 3}, abbc -> {0, 1, 2, 65538}, abca -> {0, 1,
65537, 65537}, abcb -> {0, 1, 65537, 65538}, abcc -> {0, 1, 65537, 131073},
..., acbb -> {0, 65536, 65537, 65538}, bbba -> {1, 2, 3, 3}, ..., bcaa -> {1,
65537, 65537, 65537}, ccca -> {65536, 131072, 196608, 196608}, ..., cbaa ->
{65536, 65537, 65537, 65537}};

```

In our present setting, we cut the suffix s from the cumulative integer list(s) for the word constructed so far and then select the new extension w from the pre-computed symbolic list for s . After this, the corresponding cumulative integer list for w is to be added to (the cumulative integer list of) the previously constructed word. This is done in order to detect the possible new abelian squares quickly. One function we use for selecting the new extensions is `tryProperExtensionVisList`. Next we present, as an example, one rule (of a great many) connected to it. This time the example is from the four-letter, $|s| = 12$, $|w| = 4$ case.

```

In[168]:= tryProperExtensionVisList[{0, 1, 1, 1025, 1025, 1026,
      1026, 1049 602, 1049 603, 1050 627, 1050 627, 1051 651}] =
      {bcda, bcdb, bcdc, daca, dacb, dadb, dcab, dcba, dcba, dcba, dcba};

```

These kinds of precomputed rules are quite fast to use, but, of course, a considerable amount of memory is needed to store all the structures.

The following function `generateRLthree` (or, equivalently, `generateRL`) constructs the extensions for a given word. Its arguments are states (strings that are also values of the global variable `€state`). As mentioned earlier, some of these states include "extendOrChangeRight", "extendOrChangeLeft", "testRight", "testLeft", "failBoth", and "succeeded". You can see the program for `generateRLthree` by opening the cell brackets at the end of this article. Note that even though the program was originally developed for the three-letter case (allowing short abelian repetitions of the form xx or xxx for a letter $x \in \Sigma_3 = \{a, b, c\}$), it works perfectly for the four-letter case as well in all our settings. Therefore, the more generic name `generateRL` is used for it.

Before starting the construction, we need to initialize the global (at this stage) variables depending on whether we use three or four letters. This is accomplished by using the `initialise` function that has the following main structure.

```

initialise[howManyStepsLeftAndRight_,
  givenWordForTest_, alphLetInt_: €alphThreeLetInt]

```

The variables `€extensionBoundaryLengthRight = howManyStepsLeftAndRight*€addWordLength` and `€extensionBoundaryLengthLeft = howManyStepsLeftAndRight*€addWordLength` stand for extension lengths. Both of

these lengths should be equal and of the form $k * \text{€addWordLength}$ for a positive integer k . In our examples, and in our present program, the value for €addWordLength (length of w) is equal to 4. Here is the full definition of the initialise function.

```

In[169]:= Clear[initialise];
initialise[howManyStepsLeftAndRight_,
  givenWordForTest_, alphLetInt_ : €alphThreeLetInt] :=
(€alphLetInt = alphLetInt; (* €alphLetInt needs to be
  set to €alphThreeLetInt or to €alphFourLetInt *)
If[alphLetInt == €alphThreeLetInt,
  alphThreeLetIntCase, alphFourLetIntCase];
€addWordLength = 4;
€preCheckedSuffLength = 8;
€suffLengthForWhichTryProperExtension =
  €preCheckedSuffLength - €addWordLength;
€reducedWordListSuffNAddNToExpressions =
  €reducedWordListSuff4Add4ToExpressions;
ordinalIndexForCumulIntegerListN =
  ordinalIndexForCumulIntegerList4; (* function names *)
€pointerExtendRight = 0;
€pointerExtendLeft = 0;
€extensionBoundaryLengthRight =
  howManyStepsLeftAndRight * €addWordLength;
€extensionBoundaryLengthLeft =
  howManyStepsLeftAndRight * €addWordLength;
(* Indeed, both of these should be equal and of the form
  k * €addWordLength for a positive integer k *)
€howFarExtendedRight = 0;
€howFarExtendedLeft = 0;
€indexListRight = {0};
€indexListLeft = {};
€givenWordForTest = givenWordForTest;
€cumulIntListOfGivenWordForTest =
  convertStrToCumulIntList[€givenWordForTest, €alphLetInt];
€cumulIntListReverseOfGivenWordForTest =
  convertStrToCumulIntList[
    StringReverse[€givenWordForTest], €alphLetInt]);

```

Here is an example of the three-letter case. The program consists of initialization cells and will be activated automatically.

