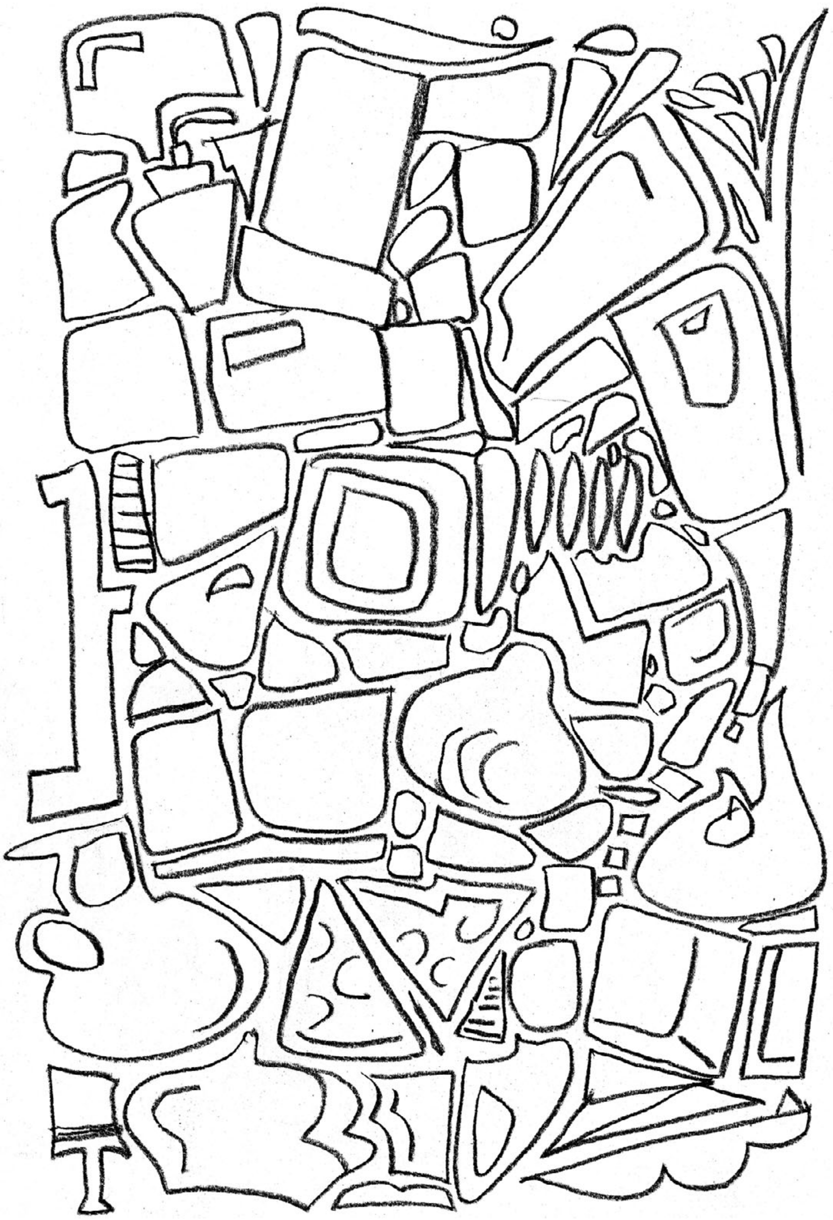
6 Text Compression



94 BW Transform of Thue–Morse Words

The goal of the problem is to show the inductive structure of the Burrows–Wheeler transform of Thue–Morse words. The words are produced by the Thue–Morse morphism μ from $\{\mathtt{a},\mathtt{b}\}^*$ to itself defined by $\mu(\mathtt{a}) = \mathtt{ab}$ and $\mu(\mathtt{b}) = \mathtt{ba}$. Iterating μ from letter a gives the nth Thue–Morse word $\tau_n = \mu^n(\mathtt{a})$ of length 2^n.

The Burrows–Wheeler transform $\mathrm{BW}(w)$ of w is the word composed of the last letters of the sorted conjugates (rotations) of w. The list of Thue–Morse words starts with $\tau_0 = \mathtt{a}$, $\tau_1 = \mathtt{ab}$, $\tau_2 = \mathtt{abba}$ and $\tau_3 = \mathtt{abbabaab}$ and the transforms of the last two are $\mathrm{BW}(\tau_2) = \mathtt{baba}$ and $\mathrm{BW}(\tau_3) = \mathtt{bbababaa}$.

Below the bar, morphism from $\{\mathtt{a},\mathtt{b}\}^*$ to itself is defined by $\overline{\mathtt{a}} = \mathtt{b}$ and $\overline{\mathtt{b}} = \mathtt{a}$.

Question. Show the Burrows–Wheeler transform $\mathrm{BW}(\tau_{n+1})$, $n > 0$, is the word $\mathtt{b}^k \cdot \overline{\mathrm{BW}(\tau_n)} \cdot \mathtt{a}^k$, where $k = 2^{n-1}$.

Solution

The solution comes from a careful inspection of the array of sorted conjugates producing the transform.

Let S_{n+1} be the $2^{n+1} \times 2^{n+1}$ array whose rows are the sorted rotations of τ_{n+1}. By definition $\mathrm{BW}(\tau_{n+1})$ is the rightmost column of S_{n+1}. The array splits into three arrays, with T_{n+1} its top 2^{n-1} rows, M_{n+1} its middle 2^n rows and B_{n+1} its bottom 2^{n-1} rows.

Example. Below are the rotations of $\tau_2 = \mathtt{abba}$ (R_2 on the left) and its sorted rotations (S_2 on the right). Thus $\mathrm{BW}(\tau_2) = \mathtt{baba}$.

$$R_2 = \begin{array}{cccc} \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} \\ \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{a} \\ \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} \\ \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{b} \end{array} \qquad S_2 = \begin{array}{cccc} \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{b} \\ \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} \\ \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} \\ \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{a} \end{array}$$

The array S_3 gives $\mathrm{BW}(\tau_3) = \mathrm{BW}(\mathtt{abbabaab}) = \mathtt{bbababaa}$.

$$S_3 = \begin{array}{cccccccc} \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} \\ \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} \\ \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} \\ \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} \\ \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} \\ \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} \\ \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} \\ \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} \end{array}$$

The decomposition of S_3 into T_3, M_3 and B_3 shows that $\mathrm{BW}(\tau_3)$ is $\mathtt{b}^2 \cdot \mathtt{abab} \cdot \mathtt{a}^2 = \mathtt{b}^2 \cdot \overline{\mathrm{BW}(\tau_2)} \cdot \mathtt{a}^2$.

$$\begin{array}{rcl}
T_3 & = & \begin{array}{cccccccc} \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} \\ \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} \\ \hline \end{array} \\
M_3 & = & \begin{array}{cccccccc} \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} \\ \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} \\ \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} \\ \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} \\ \hline \end{array} \\
B_3 & = & \begin{array}{cccccccc} \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} \\ \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} \end{array}
\end{array}$$

Since rows of S_n are sorted, a simple verification shows they remain sorted when μ is applied to them. The last column of $\mu(S_n)$ is then $\overline{\mathrm{BW}(\tau_n)}$ by the definition of μ.

It remains to find rotations of τ_{n+1} that are in T_{n+1} and in B_{n+1}, which eventually proves $M_{n+1} = \mu(S_n)$.

Observation. The number of occurrences of a's and those of b's in τ_n are both equal to 2^{n-1}.

In the word $\tau_{n+1} = \mu(\tau_n)$ let us consider the occurrences of ba that are images of an occurrence of b in τ_n. By the observation, there are 2^{n-1} such occurrences of ba. Equivalently, they start at an even position on τ_{n+1} (there are other occurrences of ba when n is large enough).

Rows of T_{n+1} are composed of rotations obtained by splitting τ_{n+1} in the middle of these factors ba. All rows of T_{n+1} start with a and end with b.

Since there is no occurrence of bbb in τ_n, the (alphabetically) greatest row of T_{n+1} cannot start with ababa and in fact starts with abaa. Thus this row is smaller than the top row of $\mu(S_n)$ that is prefixed by abab, since it is the image of a rotation of τ_n prefixed by aa.

Symmetrically, B_{n+1} is composed of rotations obtained by splitting occurrences of ab starting at an even position on τ_{n+1}. Proving they are all larger than the last row of $\mu(S_n)$ is proved similarly as above.

To conclude, since T_{n+1} and B_{n+1} each have $k = 2^{n-1}$ rows, $M_{n+1} = \mu(S_n)$. Rows of T_{n+1} end with b and provide the prefix $\mathtt{b}^k$ of $\mathrm{BW}(\tau_{n+1})$. Rows of B_{n+1} end with a and provide the suffix $\mathtt{a}^k$ of $\mathrm{BW}(\tau_{n+1})$.

95 BW Transform of Balanced Words

The Burrows–Wheeler operation maps a word w to the word $\mathrm{BW}(w)$ composed of the last letters of the sorted conjugates of w. The goal of the problem is to characterise primitive words $w \in \{\mathtt{a},\mathtt{b}\}^+$ for which $\mathrm{BW}(w) \in \mathtt{b}^+\mathtt{a}^+$. Such a word w can then be compressed to a word of length $\log|w|$ by representing the exponents of a and of b.

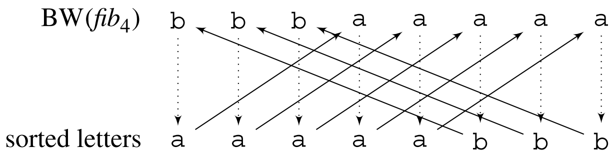
The characterisation is based on the notion of balanced words. The density (or weight) of a word $u \in \{\mathtt{a},\mathtt{b}\}^+$ is the number of occurrences of letter a in it, that is, $|u|_{\mathtt{a}}$. A word w is said to be balanced if any two factors of w, u and v of the same length have almost the same density. More formally, factors satisfy

$$|u| = |v| \Longrightarrow -1 \le |u|_{\mathtt{a}} - |v|_{\mathtt{a}} \le 1.$$

We also say the word w is circularly balanced if w^2 is balanced.

Question. For a primitive word $w \in \{\mathtt{a},\mathtt{b}\}^+$, show that w is circularly balanced if and only if $\mathrm{BW}(w) \in \mathtt{b}^+\mathtt{a}^+$.

Fibonacci words are typical examples of circularly balanced words. Below is a graph showing the cycle to recover a conjugate of fib_4 (length $F_6 = 8$ and density $F_5 = 5$), from $\mathtt{b}^3\mathtt{a}^5$. Following the cycle from the top left letter, letters of aabaabab are those met successively on the bottom line. Starting from another letter gives another conjugate of $\mathit{fib}_4 = \mathtt{abaababa}$, which itself is obtained by starting from the first occurrence of a on the top line. In fact, any word of length $|\mathit{fib}_n|$ and density $|\mathit{fib}_{n-1}|$ is circularly balanced if and only if it is a conjugate of fib_n.



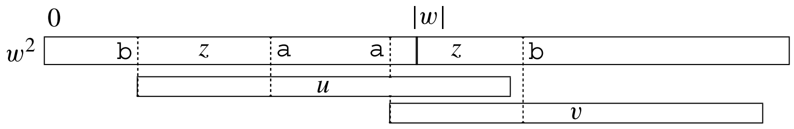
Question. Show that $\mathrm{BW}(\mathit{fib}_n) \in \mathtt{b}^+\mathtt{a}^+$ for $n > 0$.

For example, $\mathrm{BW}(\mathit{fib}_1) = \mathrm{BW}(\mathtt{ab}) = \mathtt{ba}$, $\mathrm{BW}(\mathit{fib}_2) = \mathrm{BW}(\mathtt{aba}) = \mathtt{baa}$ and $\mathrm{BW}(\mathit{fib}_3) = \mathrm{BW}(\mathtt{abaab}) = \mathtt{bbaaa}$.

Solution

Transformation of a circularly balanced word. We start with a proof of the direct implication in the first question. First note that $\mathrm{BW}(w)$, composed of letters ending lexicographically sorted factors of length $|w|$ in w^2, is equivalently composed of the letters preceding occurrences of these sorted factors starting at positions 1 to $|w|$ on w^2. The solution comes readily from the next lemma.

***Lemma* 7** *For a circularly balanced primitive word w, let* $\mathtt{b}u$ *and* $\mathtt{a}v$ *be two factors of w^2 with $|u| = |v| = |w|$. Then $u < v$.*

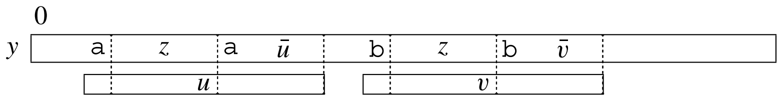


Proof Let z be the longest common prefix of u and v. Since u and v are conjugates of w and w is primitive, $u \neq v$. Thus either both $z\mathtt{a}$ is a prefix of u and $z\mathtt{b}$ is a prefix of v (like in the above picture) or both $z\mathtt{b}$ is a prefix of u and $z\mathtt{a}$ a prefix of v. But the second case is impossible because $|\mathtt{b}z\mathtt{b}| = |\mathtt{a}z\mathtt{a}|$ and $|\mathtt{b}z\mathtt{b}|_{\mathtt{a}} - |\mathtt{a}z\mathtt{a}|_{\mathtt{a}} = -2$, contradicting the balanced condition. The first case shows that $u < v$. ■

A direct consequence of the lemma is that any conjugate of w whose occurrence is preceded by $\mathtt{b}$ in w^2 is smaller than any conjugate preceded by $\mathtt{a}$. Thus $\mathrm{BW}(w) \in \mathtt{b}^+\mathtt{a}^+$.

Converse implication. To prove the converse implication we show that $\mathrm{BW}(w) \notin \mathtt{b}^+\mathtt{a}^+$ if w is not circularly balanced, which is a direct consequence of the next lemma.

***Lemma* 8** *If the primitive word $y \in \{\mathtt{a},\mathtt{b}\}^+$ is not balanced then it contains two factors of the form* $\mathtt{a}z\mathtt{a}$ *and* $\mathtt{b}z\mathtt{b}$*, for a word z.*



Proof Let u and v be factors of y of minimal length $m = |u| = |v|$ with $||u|_{\mathtt{a}} - |v|_{\mathtt{a}}| > 1$. Due to the minimality of m, u and v start with different letters, say $\mathtt{a}$ and $\mathtt{b}$ respectively. Let z be the longest common prefix of $\mathtt{a}^{-1}u$ and $\mathtt{b}^{-1}v$. The inequality $||u|_{\mathtt{a}} - |v|_{\mathtt{a}}| > 1$ implies $|z| < m - 2$. Then $u = \mathtt{a}zc\bar{u}$ and $v = \mathtt{b}zd\bar{v}$ for words $\bar{u}$ and $\bar{v}$ and for letters c and d, $c \neq d$. Due to the minimality of m again, we cannot have both $c = \mathtt{b}$ and $d = \mathtt{a}$. Then $c = \mathtt{a}$ and $d = \mathtt{b}$ (see the above picture), which shows that words $\mathtt{a}z\mathtt{a}$ and $\mathtt{b}z\mathtt{b}$ are factors of y, as expected. ■

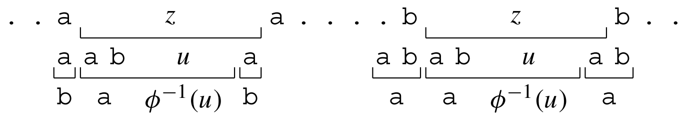
To conclude, when w is not circularly balanced, w^2 is not balanced and by the above lemma contains two factors of the forms aza and bzb. Therefore, the conjugate of w prefixed with za and preceded by a is smaller than the conjugate prefixed with zb, and preceded by b. Therefore ab is a subsequence of BW(w), which implies BW$(w) \notin \mathtt{b}^+\mathtt{a}^+$.

Case of Fibonacci words. To prove the statement of the second question, we show that Fibonacci words are circularly balanced. Since their squares are prefixes of the infinite Fibonacci word **f**, it is enough to show that the latter does not contain two factors of the forms aza and bzb for any word z. This yields the result using the conclusion of the first question.

Recall that **f** is generated by iteration of the morphism ϕ from $\{\mathtt{a},\mathtt{b}\}^*$ to itself defined by $\phi(\mathtt{a}) = \mathtt{ab}$ and $\phi(\mathtt{b}) = \mathtt{a}$: $\mathbf{f} = \phi^\infty(\mathtt{a})$. The word is also a fixed point of ϕ: $\phi(\mathbf{f}) = \mathbf{f}$.

Related to the question, note that, for example, aa is a factor of **f** but bb is not, and similarly that bab is a factor of **f** but aaa is not. That is, **f** avoids bb and aaa among many other (binary) words.

***Lemma* 9** *The infinite Fibonacci word* **f** *does not contain two factors* aza *and* bzb *for any word z.*



Proof The proof is by contradiction, assuming **f** contains two factors of the stated forms. Let z be the shortest possible word for which both aza and bzb occur in **f**.

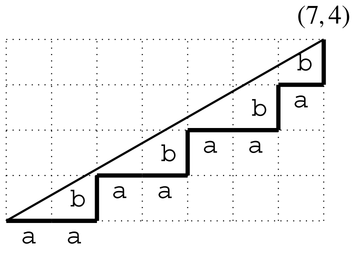
Considering words avoided by **f** like $\mathtt{a}^3$ and $\mathtt{b}^2$, it follows that z should start with ab and end with a. A simple verification shows that the length of z should be at least 4, then $z = \mathtt{ab}u\mathtt{a}$ with $u \neq \varepsilon$ (see picture). Indeed, the two occurrences of a cannot coincide because **f** avoids $\mathtt{a}^3$. Then u cannot be empty because ababab does not occur in **f**, as it would be the image by ϕ of aaa that is avoided by **f**.

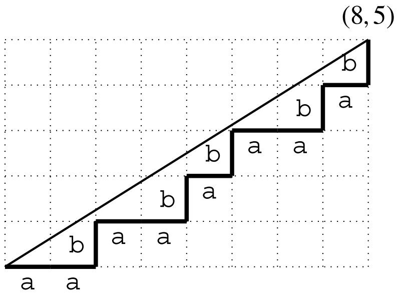
The words aab$u$a, a prefix of aza, and abab$u$ab uniquely factorise on $\{\mathtt{a},\mathtt{ab}\}$, which is a suffix code. Thus, $\phi^{-1}(\mathtt{aab}u\mathtt{a}) = \mathtt{ba}\phi^{-1}(u)\mathtt{b}$ and $\phi^{-1}(\mathtt{abab}u\mathtt{ab}) = \mathtt{aa}\phi^{-1}(u)\mathtt{a}$ occur in **f**. But this contradicts the minimality of z's length because $\mathtt{a}\phi^{-1}(u)$ is shorter than z. Therefore, **f** does not contain two factors of the forms aza and bzb, which achieves the proof. ■

Notes

The result of the problem was first shown by Mantaci et al. and appeared in a different form in [185]. Part of the present proof uses Proposition 2.1.3 in [176, chapter 2], which states additionally that the word z in the lemma of the above converse implication is a palindrome.

The question is related to Christoffel words that are balanced Lyndon words, as proved by Berstel and de Luca [33] (see also [35, 176]). The result is stated by Reutenauer in [208] as follows: let w be a Lyndon word for which $p = |w|_{\mathtt{a}}$ and $q = |w|_{\mathtt{b}}$ are relatively prime. Then w is a Christoffel word if and only if $\mathrm{BW}(w) = \mathtt{b}^q\mathtt{a}^p$.





Lower Christoffel words approximate from below segments of the plane starting from the origin. The pictures show the Christoffel word aabaabaabab (left) representing the path on grid lines closely below the segment from (0,0) to (7,4). The Lyndon word conjugate of Fibonacci word fib_5 = abaababaabaab (right) of length $F_7 = 13$ and density $F_6 = 8$ approximates the segment from (0,0) to $(F_6, F_5) = (8,5)$.

96 In-place BW Transform

The Burrows–Wheeler transform (BW) of a word can be computed in linear time, over a linear-sortable alphabet, using linear space. This is achieved via a method to sort the suffixes or conjugates of the word, which requires linear extra space in addition to the input.

The problem shows how the input word is changed to its transform with constant additional memory space but with a slower computation.

Let x be a fixed word of length n whose last letter is the end-marker #, smaller than all other letters. Sorting the conjugates of x to get BW(x) then amounts to sorting its suffixes. The transform is composed of letters preceding suffixes (circularly for the end-marker). For the example of $x =$ `banana#` we get BW(x) $=$ `annb#aa`:

BW(x)							
a	#						
n	a	#					
n	a	n	a	#			
b	a	n	a	n	a	#	
#	b	a	n	a	n	a	#
a	n	a	#				
a	n	a	n	a	#		

Question. Design an in-place algorithm to compute the Burrows–Wheeler transform of an end-marked word of length n in time $O(n^2)$ using only constant extra space.

Solution

Let initially $z = x$. The goal is to transform (the array) z in-place into BW(x). The computation is performed by scanning z right to left.

Let x_i denote the suffix $x[i\,..\,n-1]$ of x, $0 \leq i < n$. During iteration i, the word $z = x[0\,..\,i] \cdot \mathrm{BW}(x[i+1\,..\,n-1])$ is transformed into the word $x[0\,..\,i-1] \cdot \mathrm{BW}(x[i\,..\,n-1])$. To do it, letter $c = x[i]$ is processed to find the rank of x_i among the suffixes $x_i, x_{i+1}, \ldots, x_{n-1}$.

If p is the position of # on z, $p - i$ is the rank of x_{i+1} among the suffixes $x_{i+1}, x_{i+2}, \ldots, x_{n-1}$. Then $z[p]$ should be c at the end of iteration i, since it precedes the suffix x_{i+1}.

To complete the iteration it remains to locate the new position of #. Since it precedes x_i itself we have to find the rank of x_i among the suffixes $x_i, x_{i+1}, \ldots,$ x_{n-1}. This can easily be done by counting the number q of letters smaller than c in $z[i+1\,..\,n-1]$ and the number t of letters equal to c in $z[i+1\,..\,p-1]$.

Then $r = q + t$ is the sought rank of x_i. Eventually the computation consists in shifting $z[i+1\,..\,i+r]$ one position to the left in z and by setting $z[i+r]$ to #.

Example. For $x =$ `banana#` the picture below simulates the whole computation. At the beginning of iteration $i = 2$ (middle row), we have $z =$ `ban·an#a` and we process the underline letter $c =$ `n`. In `an#a` there are three letters smaller than c and, before #, one letter equal to it. Then $r = 4$. After substituting c for #, the factor $z[3\,..\,3+4-1]$ is shifted and the end marker inserted after it. This gives $z =$ `ba · anna#`.

i				x				r
	b	a	n	a	n	a	#	
4	b	a	n	a	$\underline{\texttt{n}}$	a	#	$2 = 2 + 0$
3	b	a	n	$\underline{\texttt{a}}$	a	n	#	$2 = 1 + 1$
2	b	a	$\underline{\texttt{n}}$	a	n	#	a	$4 = 3 + 1$
1	b	$\underline{\texttt{a}}$	a	n	n	a	#	$3 = 1 + 2$
0	$\underline{\texttt{b}}$	a	n	n	#	a	a	$4 = 4 + 0$
	a	n	n	b	#	a	a	

BW(x)

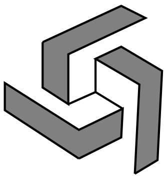
Algorithm InPlaceBW implements the above strategy. It begins with iteration $i = n - 3$, since $x[n-2\,..\,n-1]$ is its own transform.

```
InPlaceBW($x$ end-marked word of length $n$)
for $i \leftarrow n-3$ downto 0 do
    $p \leftarrow$ position of # in $x[i+1\,..\,n-1]$
    $c \leftarrow x[i]$
    $r \leftarrow 0$
    for $j \leftarrow i+1$ to $n-1$ do
        if $x[j] < c$ then
            $r \leftarrow r+1$
    for $j \leftarrow i+1$ to $p-1$ do
        if $x[j] = c$ then
            $r \leftarrow r+1$
    $x[p] \leftarrow c$
    $x[i\,..\,i+r-1] \leftarrow x[i+1\,..\,i+r]$
    $x[i+r] \leftarrow$ #
return $x$
```

As for the running time, instructions at lines 2, 5–7, 8–10 and 12 all run in time $O(n-i)$. Then the overall running time is $O(n^2)$ in the comparison model.

Notes

The material of the problem is from [73]. The authors also show how to invert in-place BW to recover the initial word with the same complexities on a constant-size alphabet. More on the Burrows–Wheeler transform is in the book on the subject by Adjeroh et al. [2].



97 Lempel–Ziv Factorisation

The problem deals with a Lempel–Ziv factorisation of words. The factorisation considered here is the decomposition of a word w into the product $w_0w_1\dots w_k$ where each w_i is the longest prefix of $w_iw_{i+1}\dots w_k$ that occurs in w before the present position $|w_0w_1\dots w_{i-1}|$. If there is no previous occurrence, w_i is the first letter of $w_iw_{i+1}\dots w_k$.

The factorisation is stored in the array LZ: LZ[0] $=$ 0 and, for $1 \le i \le k$, LZ[i] $= |w_0w_1\dots w_{i-1}|$. For example, the factorisation of `abaabababbabbb` is `a` · `b` · `a` · `aba` · `bab` · `babb` · `b`, which gives the array LZ $= [0,1,2,3,6,9,13,14]$.

Question. Show how to compute the array LZ of a word in linear time assuming a fixed-size alphabet.

The same running time can be achieved when the alphabet is linearly sortable, which is a weaker condition than the above one. This is done from the longest previous array (LPF) array of the word that computes in linear time under this condition (see Problem 53).

The LPF array of a word w is defined, for each position i on w, by: LPF[i] is the length of the longest factor of w that starts both at position i and at a smaller position. Below is the LPF array of `abaabababbabbb`.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$w[i]$	a	b	a	a	b	a	b	a	b	b	a	b	b	b
LPF[i]	**0**	**0**	**1**	**3**	2	4	**3**	2	1	**4**	3	2	2	**1**

Question. Design an algorithm that computes in linear time the Lempel–Ziv factorisation of a word given its LPF array.

Solution

Direct computation of LZ. A solution to the first question utilises the Suffix tree $T = \mathcal{ST}(w)$ of w. Its terminal nodes (or leaves if w has an end-marker) are identified with the suffixes of w and can be assumed to be labelled by their starting positions. Additionally, for each node v of T, *first*(v) is the smallest label of a leaf in the subtree rooted at v, which can be computed via a mere bottom-up traversal of the tree.

Assume LZ$[0\,..\,i-1]$ is computed and LZ$[i-1] = j$, for $1 \le i \le k$. To get LZ$[i]$ the tree is traversed from *root*(T) along the path spelling a prefix of $w[j\,..\,n-1]$ letter by letter. The descent stops if either it cannot continue or the scanned word does not occur before position j. The latter condition is checked in the following way: in a given step the current node of the tree is an explicit node v or possibly an implicit inner node, in which case we look down for the first explicit node v. Checking if a previous occurrence exists amounts to checking if *first*$(v) < j$.

Building the Suffix tree takes linear time on a linearly sortable alphabet (see Problem 47) and traversing it takes linear time on a fixed-size alphabet. It is $O(|w| \log \mathit{alph}(w))$ on a general alphabet.

LZ from LPF. The following algorithm solves the second question.

```
LZ-FACTORISATION(LPF table of a word of length $n$)
1  $(\mathrm{LZ}[0], i) \leftarrow (0,0)$
2  while $\mathrm{LZ}[i] < n$ do
3      $\mathrm{LZ}[i+1] \leftarrow \mathrm{LZ}[i] + \max\{1, \mathrm{LPF}[\mathrm{LZ}[i]]\}$
4      $i \leftarrow i+1$
5  return LZ
```

It is clear that LZ[0] is correctly set. Let us assume that, at iteration i of the while loop, values LZ$[j]$ are correct for $0 \le j \le i$. In particular, LZ$[i] = |w_0 w_1 \dots w_{i-1}|$.

Let w_i be the next factor of the factorisation. If w_i is not empty then its length (greater than 1) is LPF[$|w_0w_1 \dots w_{i-1}|$]; thus LZ[$i+1$] $=$ LZ[i] $+$ LPF[LZ[i]]. If w_i is empty then LZ[$i+1$] $=$ LZ[i] $+ 1$. In both cases, the instruction at line 3 correctly computes LZ[$i+1$].

The algorithm stops when LZ[i] $\geq n$; thus it computes all the values LZ[i] for $0 \leq i \leq k$.

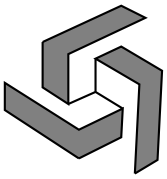
All the instructions of the algorithm run in constant time except the while loop that is iterated $k+1$ times; thus the algorithm runs in $O(k)$ time.

Notes

An alternative algorithm can be designed with the Suffix automaton (or DAWG) of the word. See [76] for the algorithm of the second question and for applications of the LPF array.

There is a large number of possible variations on the definition of the factorisation. The above version is inspired by the LZ77 compression method designed by Ziv and Lempel [243] (see [37]). Its study has been stimulated by its high performance in real applications.

The factorisation is also useful to produce efficient algorithms for locating repetitions in words (see [67, 167]), outperformed by the computation of runs in [26] (see Problem 87). The factorisation can also deal with repetitions in other applications, such as finding approximate repetitions in words [168] or aligning genomic sequences [88], for example.



98 Lempel–Ziv–Welch Decoding

The Lempel–Ziv–Welch compression method is based on a type of Lempel–Ziv factorisation. It consists in encoding repeating factors of the input text by their code in a dictionary D of words. The dictionary, initialised with all the letters of the alphabet A, is prefix-closed: every prefix of a word in the dictionary is in it.

Here is the encoding algorithm in which $code_D(w)$ is the index of the factor w in the dictionary D.

```
LZW-ENCODER(input non-empty word)
 1  $D \leftarrow A$
 2  $w \leftarrow$ first letter of input
 3  while not end of input do
 4      $a \leftarrow$ next letter of input
 5      if $wa \in D$ then
 6          $w \leftarrow wa$
 7      else WRITE($code_D(w)$)
 8          $D \leftarrow D \cup \{wa\}$
 9          $w \leftarrow a$
10  WRITE($code_D(w)$)
```

The decompression algorithm reads the sequence of codes produced by the encoder and updates the dictionary similarly to the way the encoder does.

```
LZW-DECODER(input non-empty word)
 1  $D \leftarrow A$
 2  while not end of input do
 3      $i \leftarrow$ next code of input
 4      $w \leftarrow$ factor of code $i$ in $D$
 5      WRITE($w$)
 6      $a \leftarrow$ first letter of next decoded factor
 7      $D \leftarrow D \cup \{wa\}$
```

Question. Show that during the decoding step Algorithm LZW-DECODER can read a code i that does not belong yet to the dictionary D if and only if index i corresponds to the code of aua, where au is the previous decoded factor, $a \in A$ and $u \in A^*$.

The question highlights the only critical situation encountered by the decoder. The property provides the element to ensure it can correctly decode its input.

Solution

We first prove that if just after it writes a code in the output the encoder reads $v = auaua$, with $a \in A$, $u \in A^*$, $au \in D$ and $aua \notin D$, then the decoder will read a code that does not belong to the dictionary.

The encoder starts reading $au \in D$. Then when reading the following a in v the encoder writes the code of au and adds aua to the dictionary. Going on, it reads the second occurrence of ua and writes the code of aua (since the dictionary is prefix-closed aua cannot be extended).

During the decoding step when the decoder reads the code of au, it next reads the code of aua before it is in the dictionary.

We now prove that if the decoder reads a code i that does not belong yet to the dictionary then it corresponds to the factor aua to where au is the factor corresponding to the code read just before i.

Let w be the factor corresponding to the code read just before i. The only code that has not been inserted in the dictionary before reading i corresponds to the factor wc, where c is the first letter of the factor having code i. Thus $c = w[0]$. If $w = au$ then code i corresponds to factor aua.

Example. Let the input be the word ACAGAATAGAGA over the 8-bit ASCII alphabet.

The dictionary initially contains the ASCII symbols and their indices are their ASCII codewords. It also contains an artificial end-of-word symbol of index 256.

Coding

A	C	A	G	A	A	T	A	G	A	G	A		w	written	added to D
	↑												A	65	AC, 257
		↑											C	67	CA, 258
			↑										A	65	AG, 259
				↑									G	71	GA, 260
					↑								A	65	AA, 261
						↑							A	65	AT, 262
							↑						T	84	TA, 263
								↑					A		
									↑				AG	259	AGA, 264
										↑			A		
											↑		AG		
												↑	AGA	264	
														256	

Decoding

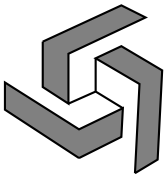
The input sequence is 65, 67, 65, 71, 65, 65, 84, 259, 264, 256.

read	written	added
65	A	
67	C	AC, 257
65	A	CA, 258
71	G	AG, 259
65	A	GA, 260
65	A	AA, 261
84	T	AT, 262
259	AG	TA, 263
264	AGA	AGA, 264
256		

The critical situation occurs when reading the index 264 because, at that moment, no word of the dictionary has this index. But since the previous decoded factor is AG, index 264 can only correspond to AGA.

Notes

The Lempel–Ziv–Welch method has been designed by Welch [239]. It improves on the initial method developed by Ziv and Lempel [243].



99 Cost of a Huffman Code

Huffman compression method applied to a text $x \in A^*$ assigns a binary codeword to each letter of x in order to produce a shortest encoded text. Its principle is that the most frequent letters are given the shortest codewords while the least frequent symbols correspond to the longest codewords.

Codewords form a prefix code (prefix-free set) naturally associated with a binary tree in which the links from a node to its left and right children are labelled by 0 and 1 respectively. Leaves correspond to original letters and labels of branches are their codewords. In the present method codes are complete: internal nodes of the tree all have exactly two children.

The cost of a Huffman code is the sum $\sum_{a \in A} \mathit{freq}(a) \times |\mathit{code}(a)|$, where $\mathit{code}(a)$ is the binary codeword of letter a. It is the smallest length of a binary text compressed by the method from a word x in which $\mathit{freq}(a) = |x|_a$ for each letter $a \in alph(x)$. Let us consider the following algorithm applied to frequencies (weights).

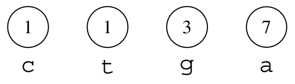
```
HuffmanCost($S$ list of positive weights)
1  $result \leftarrow 0$
2  while $|S| > 1$ do
3      $p \leftarrow$ MinDelete($S$)
4      $q \leftarrow$ MinDelete($S$)
5      add $p + q$ to $S$
6      $result \leftarrow result + p + q$
7  return $result$
```

Question. Prove that Algorithm HuffmanCost(S) computes the smallest cost of a Huffman code from a list S of item weights.

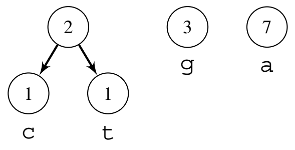
[**Hint:** Consider the Huffman tree associated with the code.]

Example. Let $S = \{7, 1, 3, 1\}$. Initially $result = 0$.
Step 1: $p = 1$, $q = 1$, $p + q = 2$, $S = \{7, 3, 2\}$, $result = 2$.
Step 2: $p = 2$, $q = 3$, $p + q = 5$, $S = \{7, 5\}$, $result = 7$.
Step 3: $p = 5$, $q = 7$, $p + q = 12$, $S = \{12\}$, $result = 19$.

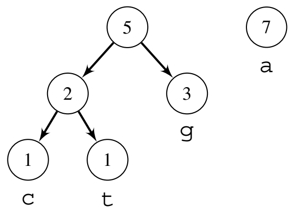
The Huffman forest underlying the algorithm, which ends up with a Huffman tree, is shown in the picture. Nodes are labelled with weights.



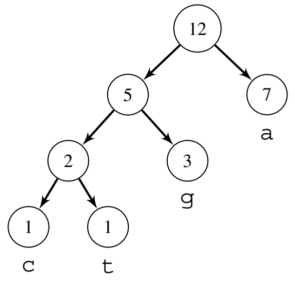
Initial state



Step 1



Step 2



Step 3

The final tree provides codewords associated with letters, summarised in the table.

	a	c	g	t
freq	7	1	3	1
code	`1`	`000`	`01`	`001`
|*code*|	1	3	2	3

The cost of the tree is $7 \times 1 + 1 \times 3 + 3 \times 2 + 1 \times 3 = 19$. It is the length of the compressed word `000 1 01 1 001 1 1 01 1 01 1 1` corresponding to `cagataagagaa`, whose letter frequencies fit with those of the example. Encoded with 8-bit codewords, the length of the latter word is 96.

Question. Show how to implement algorithm HUFFMANCOST(S) so that it runs in linear time when the list S is in increasing order.

[**Hint:** Use a queue for inserting the new values (corresponding to internal nodes of the tree).]

Solution

Correctness of HUFFMANCOST. Let S_i denote the value of S at step i of the while loop of the algorithm, $0 \leq i \leq |S| - 1$.

The loop invariant of the algorithm is: *result* is the sum of total cost of Huffman codewords representing the weights stored in S_i.

Before the first iteration, S_0 is a forest composed of node trees each of depth 0, which corresponds to the initialisation $result = 0$.

During iteration i, the algorithm selects and deletes the least two weights p and q from S_{i-1} and adds $p + q$ to S_{i-1} to produce S_i. This mimics the creation of a new tree whose root has weight $p + q$, thus creating two new edges. Then one more bit is needed to account for all the codewords of letters associated with the leaves of the new tree. Altogether this occurs $p + q$ times and implies that *result* should be incremented by $p + q$ as done at line 6. As a consequence, at the end of iteration i, *result* is the sum of the total cost of Huffman codewords representing the weights stored in S_i.

At the end of the $(|S| - 1)$th iteration only one weight is left in S and *result* is the total cost of the corresponding Huffman code.

It is clear that, at any iteration of the while loop, choosing other values than the two minimal values in S would produce a larger cost than *result*.

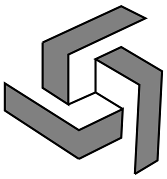
Implementation in linear time. To have HUFFMANCOST(S) running in linear time it is enough to insert newly created weights in a queue Q. Since new weights come in increasing order, Q is also sorted and each step runs in constant time, giving the following solution.

```
HuffmanCostLinear($S$ increasing list of positive weights)
1  $result \leftarrow 0$
2  $Q \leftarrow \emptyset$
3  while $|S| + |Q| > 1$ do
4      $(p,q) \leftarrow$ extract the 2 smallest values among
           the first 2 values of $S$ and the first 2 values of $Q$
5      Enqueue($Q, p+q$)
6      $result \leftarrow result + p + q$
7  return $result$
```

Example. Let $S = (1,1,3,7)$. Initially $result = 0$ and $Q = \emptyset$
Step 1: $p = 1, q = 1$, $p+q = 2$, $S = (3,7)$, $Q = (2)$, $result = 2$
Step 2: $p = 2, q = 3$, $p+q = 5$, $S = (7)$, $Q = (5)$, $result = 7$
Step 3: $p = 5, q = 7$, $p+q = 12$, $S = \emptyset$, $Q = (12)$, $result = 19$.

Notes

Huffman trees were introduced by Huffman [144]. The linear-time construction method, once the initial frequencies are already sorted, is due to Van Leeuwen [235].



100 Length-Limited Huffman Coding

Given the frequencies of alphabet letters, the Huffman algorithm builds an optimal prefix code to encode the letters in such a way that encodings are as short as possible. In the general case there is no constraint on the length of the codewords. But sometimes one may want to bound the codeword length. Building a code satisfying such a constraint is the subject of this problem.

The coin collector's problem is an example of where the constraint is used. Collectors have coins with two independent properties: denominations (currency values) and numismatic values (collector values). Their goal is to collect a sum N while minimising the total numismatic value.

Let denominations be integer powers of 2: 2^{-i} with $1 \leq i \leq L$. Coins are organised as follows: there is a list for each denomination in which coins are sorted in increasing order of their numismatic values.

The method consists in grouping adjacent coins two by two in the list of smaller denominations, dropping the last coin if their number is odd. The numismatic value of a package is the sum of numismatic values of the two coins. Newly formed packages are associated with the coins of the next smallest denomination (sorted in increasing numismatic value). The process is repeated until the list of coins of denomination 2^{-1} is processed.

Question. Design an algorithm that computes for a list of n frequencies an optimal length-limited Huffman code in which no codeword is longer than L and that runs in time $O(nL)$.

[**Hint:** Reduce the problem to the binary coin collector's problem.]

Example. A coin collector has:

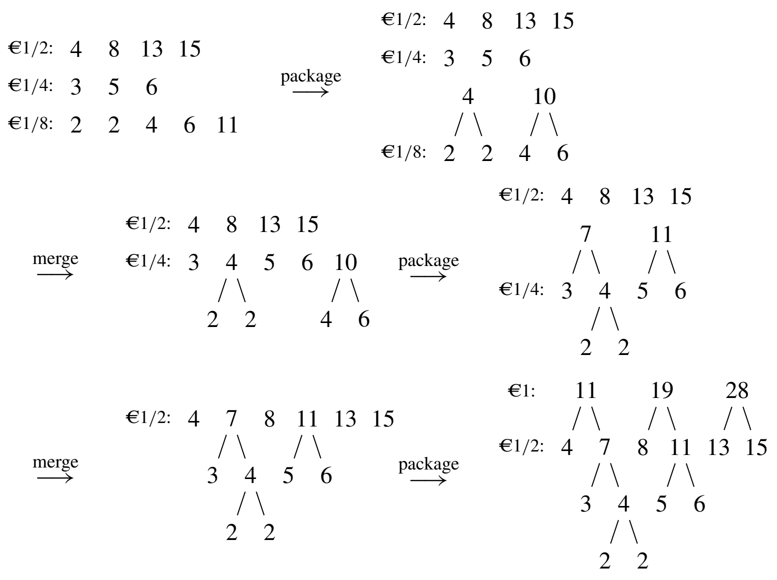
- 4 €$1/2$ coins of numismatic values 4, 8, 13 and 15 respectively,
- 3 €$1/4$ coins of numismatic values 3, 5 and 6 respectively,
- 5 €$1/8$ coins of numismatic values 2, 2, 4, 6 and 11 respectively,

and wants to collect 2 euros.

First, €$1/8$ coins are grouped two by two to form two packages of €$1/4$ with respective numismatic values 4 and 10, dropping the coin of numismatic value 11.

Then, these two packages are merged with the €$1/4$ coins and sorted. Coins and packages of €$1/4$ are grouped, which produces 2 €$1/2$ packages of respective numismatic values 7 and 11, disregarding the package of numismatic value 10.

Going on, these two packages are merged with the €1/2 coins and sorted. Finally, coins and packages of €1/2 are processed, which gives three packages of respective numismatic values 11, 19 and 28. The picture illustrates the whole process.



The first two packages give the solution: 2 euros composed of 2 €1/8 coins of numismatic values 2 each; 3 €1/4 coins of numismatic values 3, 5 and 6; and 2 €1/2 coins of numismatic values 4 and 8 for a total numismatic value of 30.

Algorithm PackageMerge(S, L) implements the strategy for a set S of coins with denominations between 2^{-L} and 2^{-1}. Package(S) groups two by two consecutive items of S and Merge(S, P) merges two sorted lists.

Eventually, the first N items of the list PackageMerge(S, L) have the lowest numismatic values and are selected to form the solution.

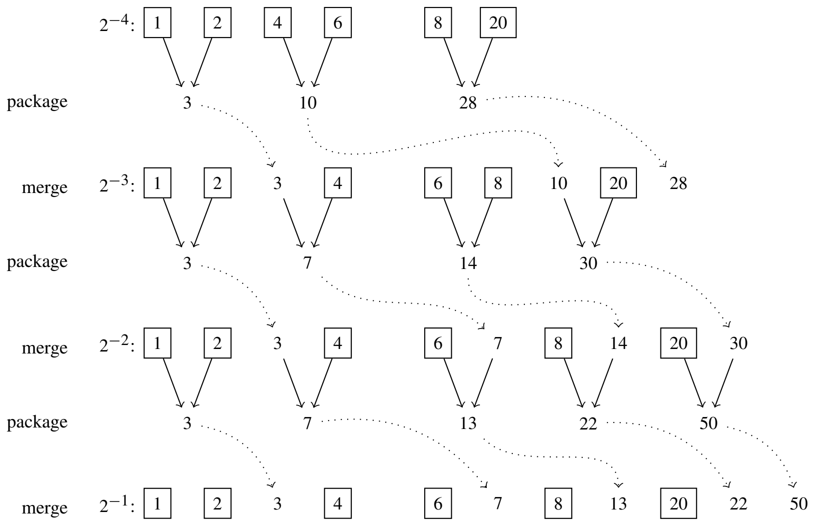
```
PackageMerge($S$ set of coins, $L$)
1  for $d \leftarrow 1$ to $L$ do
2      $S_d \leftarrow$ list of coins of $S$ with denomination $2^{-d}$
           sorted by increasing numismatic value
3  for $d \leftarrow L$ downto 1 do
4      $P \leftarrow$ Package($S_d$)
5      $S_{d-1} \leftarrow$ Merge($S_{d-1}, P$)
6  return $S_0$
```

Both PACKAGE(S') and MERGE(S', P') run in linear time according to $n = |S|$. Thus, provided that the lists of coins are already sorted, the algorithm PACKAGEMERGE(S, L) runs in time and space $O(nL)$.

Solution

Given n letter frequencies w_i for $1 \le i \le n$, the previous algorithm can be applied to collect a sum equal to $n-1$ by creating, for each $1 \le i \le n$, L coins of numismatic value w_i and of denomination 2^{-j} for each $1 \le j \le L$ to find an Huffman optimal code, where no codeword is longer than L.

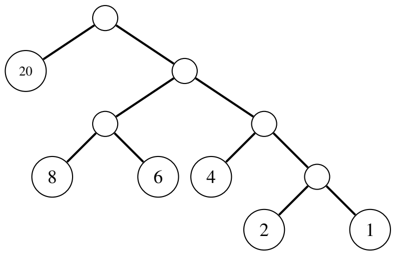
Example. Given the following six frequencies sorted in increasing order $w = (1,2,4,6,8,20)$ and $L = 4$, the PACKAGEMERGE algorithm operates as illustrated.



2^{-4}: 1 2 4 6 8 20
package 3 10 28
merge 2^{-3}: 1 2 3 4 6 8 10 20 28
package 3 7 14 30
merge 2^{-2}: 1 2 3 4 6 7 8 14 20 30
package 3 7 13 22 50
merge 2^{-1}: 1 2 3 4 6 7 8 13 20 22 50

Lengths of codewords corresponding to each frequency are computed by scanning in increasing order the first $2n - 2 = 10$ items of the last level. This is summarised in the table, where, for instance, the 6th item has weight 7 and corresponds to frequencies 1, 2 and 4. The tree corresponds to these codeword lengths.

Item	weight	1	2	4	6	8	20
1	1	1	0	0	0	0	0
2	2	1	1	0	0	0	0
3	3	2	2	0	0	0	0
4	4	2	2	1	0	0	0
5	6	2	2	1	1	0	0
6	7	3	3	2	1	0	0
7	8	3	3	2	1	1	0
8	13	4	4	3	2	1	0
9	20	4	4	3	2	1	1
10	22	4	4	3	3	3	1



More precisely, L lists of coins are considered, one for each denomination. These lists are sorted in increasing order of numismatic values. Actually, since in this case $L = O(\log n)$, sorting can be done within the given complexity and a solution can be produced in $O(nL)$ time and space complexities.

At the end, the first $2n - 2$ items of the list corresponding to 2^{-1} are processed. In these items, each occurrence of an original frequency accounts for one unit to the length of the associated codeword.

Let $(i,\ell) \in [1,n] \times [1,L]$ be a node of weight w_i and width $2^{-\ell}$. The weight (resp. width) of a set of nodes is the sum of the weights (resp. widths) of its nodes. We define *nodeset*(T) for a binary tree T with n leaves as follows: *nodeset*$(T) = \{(i,\ell) : 1 \le i \le n, 1 \le \ell \le \ell_i\}$ where ℓ_i is the depth of the ith leaf of T.

Thus the weight of *nodeset*(T) is *weight*$(T) = \sum_{i=1}^{n} w_i \ell_i$ and its width is *width*$(T) = n - 1$ (proof by induction).

***Lemma* 10** *The first $2n - 2$ items of the last list computed by Algorithm* PACKAGEMERGE *applied to L list of n coins sorted in increasing numismatic values w_i, $1 \le i \le n$ correspond to a minimum weight nodeset of width $n - 1$.*

Proof Let $C = (k_1, k_2, \ldots, k_n)$ be the codeword lengths produced by Algorithm PACKAGEMERGE. Let $K = \sum_{i=1}^{n} 2^{-k_i}$. Initially $C = (0,0,\ldots,0)$ and $K = n$. It can be easily checked that every item among the first $2n-2$ items produced by the algorithm decreases K by 2^{-1}. Thus the produced nodeset has width $n - 1$. It can also easily be checked that the algorithm at each step greedily chooses the minimal weight so that the produced nodeset is of total minimum weight. ■

A nodeset Z is monotone if the following two conditions hold:

- $(i,\ell) \in Z \Longrightarrow (i+1,\ell) \in Z$, for $1 \le i < n$,
- $(i,\ell) \in Z \Longrightarrow (i,\ell - 1) \in Z$, for $\ell > 1$.

The following lemmas can be easily proved.

***Lemma* 11** *For an integer $X < n$, the minimum weight nodeset of width X is monotone.*

***Lemma* 12** *If $(\ell_1, \ell_2, \ldots, \ell_n)$ is a list of integers for which $1 \le \ell_i \le L$ and Z is the nodeset $\{(i,\ell) : 1 \le i \le n, 1 \le \ell \le \ell_i\}$ then width$(Z) = n - \sum_{i=1}^{n} 2^{-\ell_i}$.*

***Lemma* 13** *If $y = (\ell_1, \ell_2, \ldots, \ell_n)$ is a monotone increasing list of non-negative integers whose width is equal to 1 then y is the list of depth of leaves of a binary tree.*

We can now state the main theorem.

***Theorem* 14** *If a nodeset Z has minimum weight among all nodesets of width $n-1$ then Z is the nodeset of a tree T that is an optimal solution to the length-limited Huffman coding problem.*

Proof Let Z be the minimum weight nodeset of width $n-1$. Let ℓ_i be the largest level such that $(i, \ell_i) \in A$ for each $1 \le i \le n$. By Lemma 11, Z is monotone. Thus $\ell_i \le \ell_{i+1}$ for $1 \le i < n$. Since Z is monotone and has width $n-1$, Lemma 12 implies that $\sum_{i=1}^{n} 2^{-\ell_i} = 1$. Then by Lemma 13, $(\ell_1, \ell_2, \ldots, \ell_n)$ is the list of leaf depths of a binary tree T, and hence $Z = \mathit{nodeset}(T)$.

Since Z has minimum weight among all nodesets of width $n-1$ it implies that T is optimal. ■

Notes

The coin collector's problem and the PackageMerge algorithm have been introduced by Larmore and Hirschberg in [172]. They also show that finding an optimal length-limited Huffman code can be reduced to the coin collector's problem and solved it in $O(nL)$ time and space. They further show how the space complexity can be lowered to $O(n)$. Other improvements can be found in [156] and in [220].

101 Online Huffman Coding

The two main drawbacks of the static Huffman compression method are that first, if the frequencies of letters in the source text are not known a priori, the source text has to be scanned twice and second, the Huffman coding tree must be included in the compressed file. The problem shows a solution that avoids these drawbacks.

The solution is based on a dynamic method in which the coding tree is updated each time a symbol is read from the source text. The current Huffman tree relates to the part of the text that has already been processed and evolves exactly in the same way during the decoding process.

Question. Design a Huffman compression method that reads only once the source text and does not need to store the coding tree in the compressed text.

[**Hint:** Huffman trees are characterised by the *siblings property*.]

Siblings property. Let T be a Huffman tree with n leaves (a complete binary weighted tree in which all leaves have positive weights). Nodes of T can be arranged in a list $(t_0, t_1, \ldots, t_{2n-2})$ that satisfies

- Nodes are in decreasing order of their weights: $weight(t_0) \geq weight(t_1) \geq \cdots \geq weight(t_{2n-2})$.
- For any i, $0 \leq i \leq n-2$, the consecutive nodes t_{2i} and t_{2i+1} are siblings (they have the same parent).

Solution

The encoding and decoding processes initialise the dynamic Huffman tree as a tree consisting of one node associated with an artificial symbol `ART` and whose weight is 1.

Encoding phase. During the encoding process, each time a symbol a is read from the source text, its codeword from the tree is appended to the output. However, this happens only if a appeared previously. Otherwise the code of `ART` followed by the original codeword of a is appended to the output. Afterwards, the tree is modified in the following way: first, if a is a not leaf of the tree a new node is inserted as the parent of leaf `ART` with a new leaf child labelled by a; second, the tree is updated (see below) to get a Huffman tree for the new prefix of the text.

Decoding phase. At decoding time the compressed text is parsed with the coding tree. The current node is initialised with the root corresponding to `ART` as in the encoding algorithm, and then the tree evolves symmetrically. Each time a `0` is read from the compressed file the walk down the tree follows the left link, and it follows the right link if a `1` is read. When the current node is a leaf, its associated symbol is appended to the output and the tree is updated exactly as is done during the encoding phase.

Update. During the encoding (resp. decoding) phase, when a symbol (resp. the code of) a is read, the current tree has to be updated to take into account the correct frequency of symbols. When the next symbol of the input is considered the weight of its associated leaf is incremented by 1, and the weights of ancestors have to be modified correspondingly.

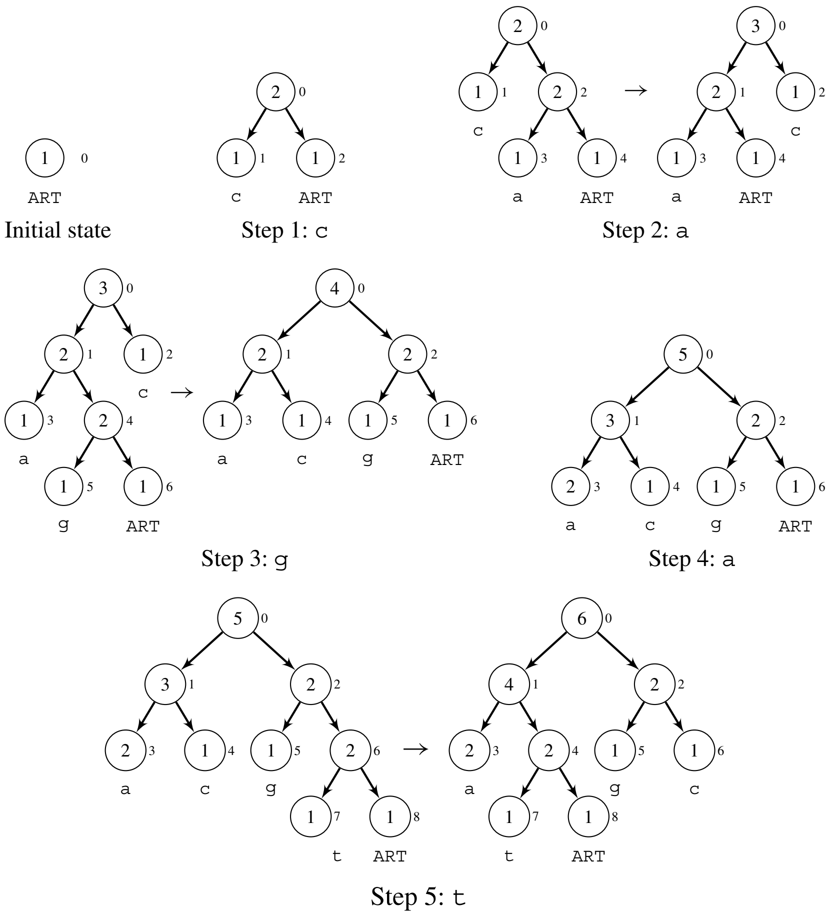
First, the weight of the leaf t_q corresponding to a is incremented by 1. Then, if the first point of the siblings property is no longer satisfied, node t_q is exchanged with the closest node t_p ($p < q$) in the list for which $weight(t_p) < weight(t_q)$. This consists in exchanging the subtrees rooted at nodes t_p and t_q. Doing so, the nodes remain in decreasing order according to their weights. Afterwards, the same operation is repeated on the parent of t_p until the root of the tree is reached.

The following algorithm implements this strategy.

```
UPDATE($a$)
1  $t_q \leftarrow leaf(a)$
2  while $t_q \neq root$ do
3      $weight(t_q) \leftarrow weight(t_q) + 1$
4      $p \leftarrow q$
5      while $weight(t_{p-1}) < weight(t_q)$ do
6          $p \leftarrow p - 1$
7      swap nodes $t_p$ and $t_q$
8      $t_q \leftarrow parent(t_p)$
9  $weight(root) \leftarrow weight(root) + 1$
```

Sketch of the proof. Assume that the siblings property holds for a Huffman tree with a list $(t_0, t_1, \ldots, t_q, \ldots, t_{2n-2})$ of nodes in decreasing order of their weights and assume that the weight of leaf t_q is incremented by 1. Then both inequalities $weight(t_p) \geq weight(t_q)$ and $weight(t_p) < weight(t_q) + 1$ imply $weight(t_p) = weight(t_q)$. Node t_p has the same weight as node t_q and thus cannot be a parent or an ancestor of t_q, since the weight of a parent is the sum of the weights of its two children and that leaves have positive weights. Then swapping t_q with the smallest node t_p such that $weight(t_p) = weight(t_q)$, incrementing *weight*($t_q$) by 1 and applying the same process to the parent of t_p up to the root restore the siblings property for the whole tree, which ensures that the tree is a Huffman tree.

The picture illustrates how the tree is updated during the first five steps of processing the input text `cagataagagaa`.



Notes

The dynamic version of the Huffman compression method presented here was discovered independently by Faller [108] and by Gallager [125]. Practical versions were given by Cormack and Horspool [62] and by Knuth [161]. A precise analysis leading to an improvement on the length of the encoding was presented by Vitter [236].

There exist myriad variants of Huffman coding; see, for example, [121].

102 Run-Length Encoding

The binary representation (expansion) of a positive integer x is denoted by $r(x) \in \{\texttt{0},\texttt{1}\}^*$. The run-length encoding of a word $w \in \texttt{1}\{\texttt{0},\texttt{1}\}^*$ is of the form $\texttt{1}^{p_0}\texttt{0}^{p_1}\cdots\texttt{1}^{p_{s-2}}\texttt{0}^{p_{s-1}}$, where $s-2 \geq 0$, p_is are positive integers for $i = 0,\ldots,s-2$ and $p_{s-1} \geq 0$. Value s is called the run length of w.

In the problem we examine the run length of the binary representation of the sum, the difference and the product of two integers.

Question. Let x and y be two integers, $x \geq y > 0$, and let n be the total run length of $r(x)$ and $r(y)$. Show that the run lengths of $r(x+y)$, $r(x-y)$ and $r(x \times y)$ are polynomial with respect to n.

Solution

Let $r(x) = \texttt{1}^{p_0}\texttt{0}^{p_1}\cdots\texttt{1}^{p_{s-2}}\texttt{0}^{p_{s-1}}$ and $r(y) = \texttt{1}^{q_0}\texttt{0}^{q_1}\cdots\texttt{1}^{q_{t-2}}\texttt{0}^{q_{t-1}}$.

Run length of $r(x+y)$**.** Let us prove by induction on n that the run length of $r(x+y)$ is polynomial w.r.t. n.

It is straightforward the induction hypothesis holds when $n = 2$, when $s = 1$ or when $t = 1$. Assume it holds when the total run length of $r(x)$ and $r(y)$ is $k < n$. Now consider the induction case when the total run length of $r(x)$ and $r(y)$ is n.

- Case $p_{s-1} \neq 0$ and $q_{t-1} \neq 0$.
  Assume w.l.o.g. that $p_{s-1} \geq q_{t-1}$. Then
  $r(x+y) = (\texttt{1}^{p_0}\texttt{0}^{p_1}\cdots\texttt{1}^{p_{s-2}}\texttt{0}^{p_{s-1}-q_{t-1}} + \texttt{1}^{q_0}\texttt{0}^{q_1}\cdots\texttt{1}^{q_{t-2}})\cdot\texttt{0}^{q_{t-1}}$.

  Since $\texttt{1}^{p_0}\texttt{0}^{p_1}\cdots\texttt{1}^{p_{s-2}}\texttt{0}^{p_{s-1}-q_{t-1}}$ and $\texttt{1}^{q_0}\texttt{0}^{q_1}\cdots\texttt{1}^{q_{t-2}}$ have total run length no more than $n-1$ by hypothesis, their sum has a run length polynomial w.r.t. n.

  Example. $r(x) = \texttt{1}^3\texttt{0}^2\texttt{1}^2\texttt{0}^3$ and $r(y) = \texttt{1}^1\texttt{0}^3\texttt{1}^3\texttt{0}^2$. Then
  $r(x+y) = (\texttt{1}^3\texttt{0}^2\texttt{1}^2\texttt{0}^1 + \texttt{1}^1\texttt{0}^3\texttt{1}^3)\cdot\texttt{0}^2 = \texttt{1}^1\texttt{0}^2\texttt{1}^1\texttt{0}^1\texttt{1}^2\texttt{0}^1\texttt{1}^1\texttt{0}^2$.

$$\begin{array}{r} \texttt{1 1 1 0 0 1 1 0 0 0} \\ +\quad \texttt{1 0 0 0 1 1 1 0 0} \\ \hline \texttt{1 0 0 1 0 1 1 0 1 0 0} \end{array} \quad = \quad \begin{array}{r} \texttt{1 1 1 0 0 1 1 0} \\ +\quad \texttt{1 0 0 0 1 1 1} \\ \hline \texttt{1 0 0 1 0 1 1 0 1}\cdot\texttt{0 0} \end{array}$$

- Case $p_{s-1} = 0$ and $q_{t-1} \neq 0$.
  If $p_{s-2} \geq q_{t-1}$ then
  $r(x+y) = (\texttt{1}^{p_0}\texttt{0}^{p_1}\cdots\texttt{1}^{p_{s-2}-q_{t-1}} + \texttt{1}^{q_0}\texttt{0}^{q_1}\cdots\texttt{1}^{q_{t-2}})\cdot\texttt{1}^{q_{t-1}}$.

Since $1^{p_0}0^{p_1}\cdots 1^{p_{s-2}-q_{t-1}}$ and $1^{q_0}0^{q_1}\cdots 1^{q_{t-2}}$ have total run length no more than $n-1$ by hypothesis, their sum has run length polynomial w.r.t. n.

Example. $r(x) = 1^3 0^2 1^5 0^0$ and $r(y) = 1^1 0^3 1^3 0^2$. Then
$r(x+y) = (1^3 0^2 1^3 + 1^1 0^3 1^3) \cdot 1^2 = 1^1 0^2 1^1 0^1 1^3 0^1 1^2$.

$$\begin{array}{r} 1\;1\;1\;0\;0\;1\;1\;1\;1\;1 \\ +\quad 1\;0\;0\;0\;1\;1\;1\;0\;0 \\ \hline 1\;0\;0\;1\;0\;1\;1\;1\;0\;1\;1 \end{array} \quad = \quad \begin{array}{r} 1\;1\;1\;0\;0\;1\;1\;1 \\ +\quad 1\;0\;0\;0\;1\;1\;1 \\ \hline 1\;0\;0\;1\;0\;1\;1\;1\;0\cdot 1\;1 \end{array}$$

If $p_{s-2} < q_{t-1}$ then
$r(x+y) = (1^{p_0}0^{p_1}\cdots 0^{p_{s-3}} + 1^{q_0}0^{q_1}\cdots 1^{q_{t-2}}0^{q_{t-1}-p_{s-2}}) \cdot 1^{p_{s-2}}$.

Since $1^{p_0}0^{p_1}\cdots 0^{p_{s-3}}$ and $1^{q_0}0^{q_1}\cdots 1^{q_{t-2}}0^{q_{t-1}-p_{s-2}}$ have total run length no more than $n-1$ by hypothesis, their sum has run length polynomial w.r.t. n.

Example. $r(x) = 1^3 0^2 1^3 0^1 1^1 0^0$ and $r(y) = 1^1 0^3 1^3 0^2$. Then
$r(x+y) = (1^3 0^2 1^3 0^1 + 1^1 0^3 1^3 0^1) \cdot 1^1 = 1^1 0^2 1^1 0^1 1^3 0^2 1^1$.

$$\begin{array}{r} 1\;1\;1\;0\;0\;1\;1\;1\;0\;1 \\ +\quad 1\;0\;0\;0\;1\;1\;1\;0\;0 \\ \hline 1\;0\;0\;1\;0\;1\;1\;1\;0\;0\;1 \end{array} \quad = \quad \begin{array}{r} 1\;1\;1\;0\;0\;1\;1\;1\;0 \\ +\quad 1\;0\;0\;0\;1\;1\;1\;0 \\ \hline 1\;0\;0\;1\;0\;1\;1\;1\;0\;0\cdot 1 \end{array}$$

- The case where $p_{s-1} \neq 0$ and $q_{t-1} = 0$ can be dealt with similarly.
- Case $p_{s-1} = 0$ and $q_{t-1} = 0$.

  Then assume w.l.o.g. that $p_{s-2} \geq q_{t-2}$. Then
  $r(x+y) = (1^{p_0}0^{p_1}\cdots 1^{p_{s-2}-q_{t-2}} + 1^{q_0}0^{q_1}\cdots 0^{q_{t-3}} + 1) \cdot 1^{q_{t-2}-1}0$.

  Since $1^{p_0}0^{p_1}\cdots 1^{p_{s-2}-q_{t-2}}$ and $1^{q_0}0^{q_1}\cdots 0^{q_{t-3}}$ have total run length no more than $n-1$ by hypothesis, their sum has run length polynomial w.r.t. n.

Example. $r(x) = 1^3 0^2 1^5 0^0$ and $r(y) = 1^1 0^5 1^3 0^0$. Then
$r(x+y) = (1^3 0^2 1^2 + 1^1 0^5 + 1) \cdot 1^2 0^1 = 1^1 0^2 1^1 0^1 1^1 0^2 1^2 0^1$.

$$\begin{array}{r} 1\;1\;1\;0\;0\;1\;1\;1\;1\;1 \\ +\quad 1\;0\;0\;0\;0\;0\;1\;1\;1 \\ \hline 1\;0\;0\;1\;0\;1\;0\;0\;1\;1\;0 \end{array} \quad = \quad \begin{array}{r} 1\;1\;1\;0\;0\;1\;1 \\ +\quad 1\;0\;0\;0\;0\;0 \\ +\quad 1 \\ \hline 1\;0\;0\;1\;0\;1\;0\;0\cdot 1\;1\;0 \end{array}$$

The conclusion of all cases answers the question for $r(x+y)$.

Run length of $r(x-y)$**.** We prove by induction on n that the run length $r(x-y)$ is polynomial w.r.t. n.

The induction hypothesis obviously holds when $n = 2$. Assume it holds when the total run length of $r(x)$ and $r(y)$ is equal to $k < n$. Consider x and y whose total run length of $r(x)$ and $r(y)$ is n.

- Case $p_{s-1} \neq 0$ and $q_{t-1} \neq 0$.

  Assume w.l.o.g. that $p_{s-1} \geq q_{t-1}$. Then
  $r(x-y) = (1^{p_0}0^{p_1}\cdots 1^{p_{s-2}}0^{p_{s-1}-q_{t-1}} + 1^{q_0}0^{q_1}\cdots 1^{q_{t-2}})\cdot 0^{q_{t-1}}$.

  Since $1^{p_0}0^{p_1}\cdots 1^{p_{s-2}}0^{p_{s-1}-q_{t-1}}$ and $1^{q_0}0^{q_1}\cdots 1^{q_{t-2}}$ have total run length no more than $n-1$ by hypothesis, their difference has run length polynomial w.r.t. n.

  Example. $r(x) = 1^3 0^2 1^2 0^3$ and $r(y) = 1^1 0^3 1^3 0^2$. Then
  $r(x-y) = (1^3 0^2 1^2 0^1 - 1^1 0^3 1^3)\cdot 0^2 = 1^1 0^2 1^5 0^2$.

$$\begin{array}{r} 1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 0 \\ -\quad 1\ 0\ 0\ 0\ 1\ 1\ 1\ 0\ 0 \\ \hline 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \end{array} = \begin{array}{r} 1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\quad\ \ \\ -\quad 1\ 0\ 0\ 0\ 1\ 1\ 1\quad\ \ \\ \hline 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\cdot 0\ 0 \end{array}$$

- Case $p_{s-1} = 0$ and $q_{t-1} \neq 0$.

  If $p_{s-2} \geq q_{t-1}$ then
  $r(x-y) = (1^{p_0}0^{p_1}\cdots 1^{p_{s-2}-q_{t-1}} + 1^{q_0}0^{q_1}\cdots 1^{q_{t-2}})\cdot 1^{q_{t-1}}$.

  Since $1^{p_0}0^{p_1}\cdots 1^{p_{s-2}-q_{t-1}}$ and $1^{q_0}0^{q_1}\cdots 1^{q_{t-2}}$ have total run length no more than $n-1$ by hypothesis, their difference has run length polynomial w.r.t. n.

  Example. $r(x) = 1^3 0^2 1^5 0^0$ and $r(y) = 1^1 0^3 1^3 0^2$. Then
  $r(x-y) = (1^3 0^2 1^3 - 1^1 0^3 1^3)\cdot 1^2 = 1^1 0^1 1^1 0^5 1^2$.

$$\begin{array}{r} 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1 \\ -\quad 1\ 0\ 0\ 0\ 1\ 1\ 1\ 0\ 0 \\ \hline 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 1 \end{array} = \begin{array}{r} 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\quad\ \ \\ -\quad 1\ 0\ 0\ 0\ 1\ 1\ 1\quad\ \ \\ \hline 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0\cdot 1\ 1 \end{array}$$

  If $p_{s-2} < q_{t-1}$ then
  $r(x-y) = (1^{p_0}0^{p_1}\cdots 0^{p_{s-3}} - 1^{q_0}0^{q_1}\cdots 1^{q_{t-2}}0^{q_{t-1}-p_{s-2}})\cdot 1^{q_{t-1}}$.

  Since $1^{p_0}0^{p_1}\cdots 0^{p_{t-3}}$ and $1^{q_0}0^{q_1}\cdots 1^{q_{t-2}}0^{q_{t-1}-p_{d-2}}$ have total run length no more than $n-1$ by hypothesis, their difference has run length polynomial w.r.t. n.

  Example. $r(x) = 1^3 0^2 1^2 0^1 1^2 0^0$ and $r(y) = 1^1 0^3 1^2 0^3$. Then
  $r(x-y) = (1^3 0^2 1^2 0^1 - 1^1 0^3 1^2 0^1)\cdot 1^2 = 1^1 0^1 1^1 0^5 1^2$.

$$\begin{array}{r} 1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ 1 \\ -\quad 1\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 0 \\ \hline 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 1 \end{array} = \begin{array}{r} 1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\quad\ \ \\ -\quad 1\ 0\ 0\ 0\ 1\ 1\ 0\quad\ \ \\ \hline 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0\cdot 1\ 1 \end{array}$$

- The case where $p_{s-1} \neq 0$ and $q_{t-1} = 0$ can be dealt with similarly.
- Case $p_{s-1} = 0$ and $q_{t-1} = 0$.

If $p_{s-2} \geq q_{t-2}$. Then
$r(x-y) = (1^{p_0}0^{p_1}\cdots 1^{p_{s-2}-q_{t-2}} - 1^{q_0}0^{q_1}\cdots 0^{q_{t-3}}) \cdot 0^{q_{t-2}}$.

Since $1^{p_0}0^{p_1}\cdots 1^{p_{s-2}-q_{t-2}}$ and $1^{q_0}0^{q_1}\cdots 0^{q_{t-3}}$ have total run length no more than $n-1$ by hypothesis, their difference has run length polynomial w.r.t. n.

Example. $r(x) = 1^3 0^2 1^5 0^0$ and $r(y) = 1^1 0^3 1^2 0^1 1^2 0^0$. Then
$r(x-y) = (1^3 0^2 1^3 - 1^1 0^3 1^2 0^1) \cdot 0^2 = 1^1 0^1 1^1 0^5 1^2$.

$$\begin{array}{ccccccccccc} & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ - & & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ \hline & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array} = \begin{array}{cccccccccccc} & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & & & \\ - & & 1 & 0 & 0 & 0 & 1 & 1 & 0 & & & \\ \hline & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & \cdot & 0 & 0 \end{array}$$

If $p_{s-2} < q_{t-2}$. Then
$r(x-y) = (1^{p_0}0^{p_1}\cdots 0^{p_{s-3}} - 1^{q_0}0^{q_1}\cdots 1^{q_{t-2}-p_{s-2}}) \cdot 0^{q_{t-2}}$.

Since $r(x-y) = (1^{p_0}0^{p_1}\cdots 0^{p_{s-3}}$ and $1^{q_0}0^{q_1}\cdots 1^{q_{t-2}-p_{s-2}}$ have total run length no more than $n-1$ by hypothesis, their difference has run length polynomial w.r.t. n.

Example. $r(x) = 1^3 0^2 1^1 0^1 1^3 0^0$ and $r(y) = 1^1 0^3 1^5 0^0$. Then
$r(x-y) = (1^3 0^2 1^1 0^1 - 1^1 0^3 1^2) \cdot 0^3 = 1^1 0^2 1^4 0^3$.

$$\begin{array}{ccccccccccc} & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ - & & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ \hline & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{array} = \begin{array}{cccccccccccc} & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & & & \\ - & & 1 & 0 & 0 & 0 & 1 & 1 & 1 & & & \\ \hline & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & \cdot & 0 & 0 \end{array}$$

The conclusion of all cases answers the question for $r(x-y)$.

Run length of $r(x \times y)$. Let us prove by induction on n that the run length of $r(x \times y)$ is polynomial w.r.t. n.

The conclusion readily comes when $n = 2$. Let us assume that the induction hypothesis holds when $r(x)$ and $r(y)$ have total run length $k < n$. Consider $r(x)$ and $r(y)$ whose total run length is n.

- Case $p_{s-1} \neq 0$. Then
  $r(x \times y) = (1^{p_0}0^{p_1}\cdots 1^{p_{s-2}} \times 1^{q_0}0^{q_1}\cdots 1^{q_{t-2}}0^{q_{t-1}}) \cdot 0^{p_{s-1}}$.

  Since $1^{p_0}0^{p_1}\cdots 1^{p_{s-2}}$ and $1^{q_0}0^{q_1}\cdots 1^{q_{t-2}}0^{q_{t-1}}$ have total run length no more than $n-1$ by hypothesis, their product has run length polynomial w.r.t. n.

  Example. $r(x) = 1^3 0^2 1^2 0^3$ and $r(y) = 1^1 0^3 1^5 0^0$. Then
  $r(x \times y) = (1^3 0^2 1^2 \times 1^1 0^3 1^5) \cdot 0^3$.

$$\begin{array}{r} 1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 0 \\ \times\quad 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1 \\ \hline \end{array} \quad = \quad \begin{array}{r} 1\ 1\ 1\ 0\ 0\ 1\ 1 \\ \times\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1 \\ \hline \cdot\ 0\ 0\ 0 \end{array}$$

- The case when $q_{t-1} \neq 0$ can be dealt with similarly.
- Case $p_{s-1} = 0$ and $q_{t-1} = 0$. Then $r(x \times y)$ is $(1^{p_0}0^{p_1} \cdots 1^{p_{s-2}} \times 1^{q_0}0^{q_1} \cdots 0^{q_{t-3}+q_{t-2}}) + (1^{p_0}0^{p_1} \cdots 1^{p_{s-2}} \times 1^{q_{t-2}})$.

  Since $1^{p_0}0^{p_1} \cdots 1^{p_{s-2}}$ and $1^{q_0}0^{q_1} \cdots 0^{q_{t-3}+q_{t-2}}$ have total run length no more than $n-1$ by hypothesis, their product has run length polynomial w.r.t. n. And since $1^{p_0}0^{p_1} \cdots 1^{p_{s-2}}$ and $1^{q_{t-2}}$ have total run length less than n by hypothesis, their product has run length polynomial w.r.t. n.

  Example. $r(x) = 1^3 0^2 1^2 0^0$ and $r(y) = 1^1 0^2 1^3 0^0$. Then $r(x \times y) = (1^3 0^2 1^2 \times 1^1 0^5) + (1^3 0^2 1^2 \times 1^3)$.

$$\begin{array}{r} 1\ 1\ 1\ 0\ 0\ 1\ 1 \\ \times\quad 1\ 0\ 0\ 1\ 1\ 1 \end{array} \quad = \quad \begin{array}{r} 1\ 1\ 1\ 0\ 0\ 1\ 1 \\ \times\quad 1\ 0\ 0\ 0\ 0\ 0 \end{array} \quad + \quad \begin{array}{r} 1\ 1\ 1\ 0\ 0\ 1\ 1 \\ \times\qquad\quad 1\ 1\ 1 \end{array}$$

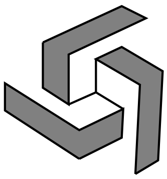
The conclusion of all cases answers the question for $r(x \times y)$.

Notes

We can also consider arithmetic operations on succinct representations on numbers in the decimal numeration system. For example,

$$1^{5n} / 41 \;=\; 271\,(00271)^{n-1}.$$

However, it is not a run-length encoding but rather its extension.



103 A Compact Factor Automaton

A Factor automaton is a minimal (deterministic) automaton accepting all the factors of a word. It is also called a directed acyclic word graph (DAWG). All its states are terminal and its edges are labelled by single letters. For certain well-structured words the automaton can be highly compressed by removing nodes having exactly one parent and one child and labelling edges by factors of the word accordingly. The resulting DAWG is called a compact DAWG (CDAWG) or compact Suffix automaton (CSA) if nodes corresponding to suffixes are marked as terminal.

The problem considers words whose CDAWGs are extremely small, namely Fibonacci words fib_n and their shortened version g_n. The word g_n is fib_n with the last two letters deleted, that is, $g_n = fib_n\{\mathtt{a},\mathtt{b}\}^{-2}$.

Question. Describe the structure of CDAWGs of Fibonacci words fib_n and of their trimmed versions g_n. Using this structure compute the number of distinct factors occurring in the words.

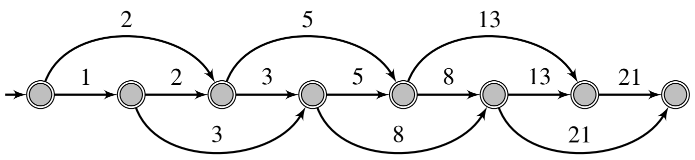
Solution

The solution is based on the lazy Fibonacci numeration system that uses the fact that each integer $x \in [1\,..\,F_n - 2]$, $n \geq 4$, is uniquely represented as $x = F_{i_0} + F_{i_1} + \cdots + F_{i_k}$, where $(F_{i_t} : 2 \leq i_t \leq n-2)$ is an increasing sequence of Fibonacci numbers satisfying

$$(*)\ i_0 \in \{0,1\} \text{ and } i_t \in \{i_{t-1}+1, i_{t-1}+2\} \text{ for } t > 0.$$

For the example of $n = 8$ the sequence of indices $(3,4,6)$ corresponds to 13 because $F_3 + F_4 + F_6 = 2 + 3 + 8 = 13$.

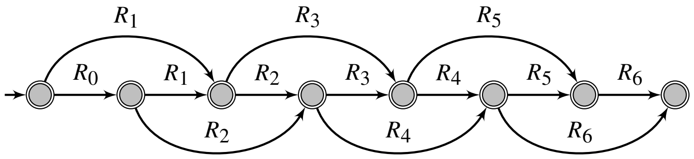
The set of sequences $(F_{i_t} : 2 \leq i_t \leq n-2)$ satisfying $(*)$ is accepted by a simple deterministic acyclic automaton whose edge labels are Fibonacci numbers and all states are terminal. The picture displays the case $n = 10$ for integers in $[1\,..\,53]$.



CDAWG of trimmed Fibonacci words. The previous automaton can be easily transformed into the CDAWG of g_n using the following property (introduced in Problem 56). Let R_i denote the reverse of fib_i and let $\mathit{suf}(k,n)$ be the kth suffix $g_n[k\,..\,|g_n|-1]$ of g_n.

Property. For $n > 2$, $\mathit{suf}(k,n)$ uniquely factorises as $R_{i_0}R_{i_1}\cdots R_{i_m}$, where $i_0 \in \{0,1\}$ and $i_t \in \{i_{t-1}+1, i_{t-1}+2\}$ for $t > 0$.

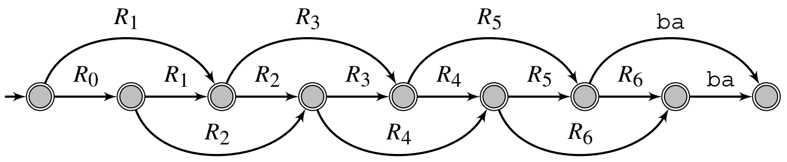
With the help of the property the previous automaton is changed into CDAWG(g_n) by substituting R_i for each Fibonacci number F_i. The next picture shows CDAWG(g_{10}) after the above picture.



Counting factors in trimmed Fibonacci words. A CDAWG is useful to compute the number of (distinct) non-empty factors occurring in the corresponding word. Indeed, it is the sum of edge lengths multiplied by the number of paths that contain these edges. The number is in fact of $F_{n-1}F_{n-2}-1$ factors in g_n, that we get using the formula, for $n > 2$: $F_2^2+F_3^2+\cdots+F_{n-2}^2 = F_{n-1}F_{n-2}-1$.

For the example of CDAWG(g_{10}) in the above picture we obtain $1^2+2^2+3^2+5^2+8^2+13^2+21^2 = 21 \times 34 - 1 = 713$ non-empty factors.

CDAWG of Fibonacci words. The compacted DAWG of Fibonacci word fib_n differs only slightly from that of the trimmed Fibonacci word g_n. We just need to append the last two trimmed letters, which lead to a simple modification of CDAWG(g_n), as done in the next picture to get CDAWG(fib_{10}). The compacted structure represents all the 781 factors of fib_{10}.



The number of factors in the Fibonacci word fib_n is slightly larger than their number in g_n since we have to consider two additional letters on the two edges reaching the last node. For $n > 2$, fib_n contains $F_{n-1}F_{n-2}+2F_{n-1}-1$ non-empty factors.

In the example $n = 10$, the additional word is ba. It is on 34 paths ending in the last node, so we have to add $2 \cdot 34 = 68$ factors. Hence fib_{10} contains $713 + 68 = 781$ non-empty factors.

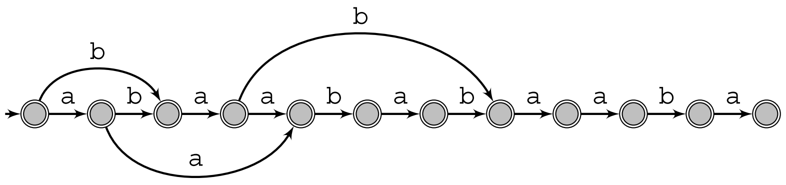
Notes

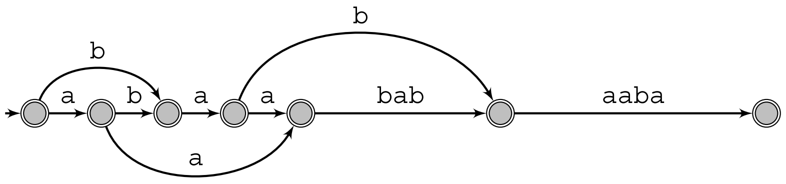
The structure of CDAWGs of Fibonacci words is described in [213]. Other very compressed and useful DAWGs appear in the more general context of Sturmian words; see [27]. The number of nodes in the structures reflects the amount of memory space to store them because labels can be represented by pairs of indices on the underlying word.

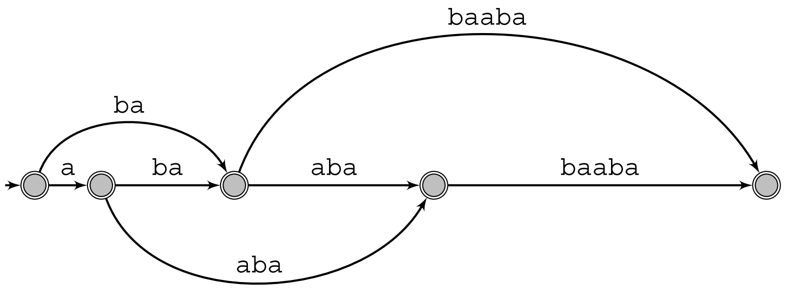
The Suffix or Factor automaton of a word of length ℓ has at least $\ell+1$ states. In fact on the binary alphabet the lower bound is achieved only when the word is a prefix of a Sturmian word, which a Fibonacci word is [221].

As mentioned at the beginning the simplest strategy to compact a DAWG is to delete nodes with unique predecessor and successor (see [38, 101, 150]). The above method for Fibonacci factors not only gives a smaller CDAWG but also provides a more useful structure.

Below are the Suffix automaton of g_7 of length 11 with 12 states, its ordinary compact version with 7 nodes and the compact version from the above technique with only 5 nodes.







Finite Thue–Morse words have similarly a very short description, see [204], from which one can easily derive that the number of factors in the Thue–Morse word of length $n \geq 16$ is $\frac{73}{192}n^2 + \frac{8}{3}$.

104 Compressed Matching in a Fibonacci Word

Compressed matching refers to the following problem: given compact representations of a pattern and of a text, locate the pattern in the text in a fast manner according to their compressed representations. The representation sizes can be logarithmic with respect to the real input sizes, as it takes place in this problem example.

The input depends on the type of compressed representation. Here we consider a very simple case where the pattern is specified as a concatenation of Fibonacci words and its representation is the sequence of their indices. The searched text is the infinite Fibonacci word $\mathbf{f} = \phi^{\infty}(\mathtt{a})$, where ϕ is the morphism defined by $\phi(\mathtt{a}) = \mathtt{ab}$ and $\phi(\mathtt{b}) = \mathtt{a}$.

The word b is added with index -1 to the list of Fibonacci words: $fib_{-1} = \mathtt{b}, fib_0 = \mathtt{a}, fib_1 = \mathtt{ab}, fib_2 = \mathtt{aba}, fib_3 = \mathtt{abaab}, \ldots$

Question. Given a sequence of integers $k_1, k_2, \ldots, k_n$ ($k_i \geq -1$) show how to check in time $O(n + k_1 + k_2 + \cdots + k_n)$ if $fib_{k_1} fib_{k_2} \cdots fib_{k_n}$ occurs in the infinite Fibonacci word $\mathbf{f}$.

Solution

The algorithm input is the sequence $w = (k_1, k_2, \ldots, k_n)$ of indices of Fibonacci words. Let $first(w)$ and $last(w)$ denote the first and last elements of w respectively.

```
CompressedMatch($w$ sequence of indices $\geq -1$)
 1  while $|w| > 1$ do
 2      if $w$ contains a factor $(i, -1), i \notin \{0, 2\}$ then
 3          return FALSE
 4      if $first(w) = -1$ then
 5          $first(w) \leftarrow 1$
 6      if $last(w) = 2$ then
 7          $last(w) \leftarrow 1$
 8      if $last(w) = 0$ then
 9          remove the last element
10      change all factors $(0, -1)$ of $w$ to 1
11      change all factors $(2, -1)$ of $w$ to $(1, 1)$
12      decrease all elements of $w$ by 1
13  return TRUE
```

Example. With the input sequence $w = (0,1,3,0,1,4)$ the algorithm makes five iterations and returns TRUE:
$(0,1,3,0,1,4) \to (-1,0,2,-1,0,3) \to (0,-1,0,0,-1,2)$
$\to (0,-1,0,0) \to (0,-1) \to (0)$.

Algorithm CompressedMatch simply implements the following observations on the sequence $w = (k_1, k_2, \dots, k_n)$.

Case (i): $\mathit{fib}_i\mathit{fib}_{-1}$ is a factor of $\mathbf{f}$ if and only if $i = 0$ or $i = 2$. Indeed, if $i = 0$, $\mathit{fib}_i\mathit{fib}_{-1} = \mathtt{ab}$, if $i = 2$, $\mathit{fib}_i\mathit{fib}_{-1} = \mathtt{abab}$, and both words appear in $\mathbf{f}$. Otherwise $\mathit{fib}_i\mathit{fib}_{-1}$ has a suffix $\mathtt{bb}$ or $\mathtt{ababab} = \phi(\mathtt{aaa})$ that does not appear in $\mathbf{f}$.

Case (ii): If $k_1 = -1$ it can be changed to 1 because the first letter b should be preceded by a in $\mathbf{f}$ and $\mathit{fib}_1 = \mathtt{ab}$.

Case (iii): Similarly, if $k_n = 2$ it can be changed to 1 because in $\mathbf{f}$ each occurrence of b is also followed by a, so the suffix aba can be reduced to ab without changing the final output.

Case (iv): Factor $(0,-1)$ can be replaced by 1 since $\mathit{fib}_0\mathit{fib}_{-1} = \mathtt{ab} = \mathit{fib}_1$ and factor $(2,-1)$ by $(1,1)$ since $\mathit{fib}_2\mathit{fib}_{-1} = \mathtt{abab} = \mathit{fib}_1\mathit{fib}_1$.

Case (v): The only problematic case is when $k_n = 0$. This corresponds to a match with an occurrence of a in $\mathbf{f}$. The correctness proof of this case follows again from the fact that the letter a occurs after each occurrence of b in $\mathbf{f}$. There are two subcases depending on the last letter of the penultimate Fibonacci factor.

Case $\mathit{fib}_{k_{n-1}}$ ends with b: Then $\mathit{fib}_{k_1}\mathit{fib}_{k_2}\cdots\mathit{fib}_{k_n}$ occurs in $\mathbf{f}$ if and only if $\mathit{fib}_{k_1}\mathit{fib}_{k_2}\cdots\mathit{fib}_{k_{n-1}}$ does occur in $\mathbf{f}$, since each occurrence of b is followed by a. Hence the last a is redundant and can be removed.

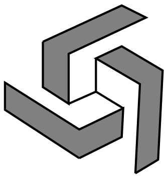
Case $\mathit{fib}_{k_{n-1}}$ ends with a: After 0 is removed and line 12 is executed the algorithm checks if $\mathit{fib}_{k_1-1}\mathit{fib}_{k_2-1}\cdots\mathit{fib}_{k_{n-1}-1}$ occurs in $\mathbf{f}$. However, $\mathit{fib}_{k_{n-1}-1}$ now ends with b and $\mathit{fib}_{k_1-1}\mathit{fib}_{k_2-1}\cdots\mathit{fib}_{k_{n-1}-1}$ occurs in $\mathbf{f}$ if and only if $v = \mathit{fib}_{k_1-1}\mathit{fib}_{k_2-1}\cdots\mathit{fib}_{k_{n-1}-1}\mathtt{a}$ does. Hence if v occurs in $\mathbf{f}$ the word $\mathit{fib}_{k_1}\mathit{fib}_{k_2}\cdots\mathit{fib}_{k_n}$ occurs in $\phi(v)$. This shows that the last element $k_n = 0$ can be removed without spoiling the correctness.

Note that when the algorithm executes the instruction at line 12 all indices of w are non-negative. Thus the argument just considered also applies after execution. This ends the proof of correctness.

As for the complexity, observe that each pair of consecutive iterations decreases the sum of indices by at least 1. Consequently the algorithm achieves the required running time of $O(n + k_1 + k_2 + \cdots + k_n)$.

Notes

The present algorithm has been proposed by Rytter as a problem for the Polish Competition in Informatics. An alternative and completely different algorithm can be found in [238]. Yet another algorithm can be obtained using compressed factor graphs of Fibonacci words.



105 Prediction by Partial Matching

Prediction by Partial Matching (PPM) is a lossless compression technique where the encoder maintains a statistical model of the text. The goal is to predict letters following a given factor of the input. In the problem we examine the data structure used to store the model.

Let y be the text to be compressed and assume $y[0\,..\,i]$ has already been encoded. PPM assigns a probability $p(a)$ to each letter $a \in A$ depending on the number of occurrences of $y[i+1-d\,..\,i] \cdot a$ in $y[0\,..\,i]$, where d is the context length. Then PPM sends $p(y[i+1])$ to an adaptive arithmetic encoder taking into account letter probabilities. When there is no occurrence of $y[i+1-d\,..\,i+1]$ in $y[0\,..\,i]$ the encoder decrements the value of d until either an occurrence of $y[i+1-d\,..\,i+1]$ is found or $d=-1$. In the last case, $y[i+1]$ is a letter not met before. Each time the encoder decrements the value of d it sends a so-called 'escape' probability to the adaptive arithmetic encoder.

PPM* is a variant of PPM which does not consider a maximum context length but stores all contexts. The initial context at each step is the shortest deterministic context, one that corresponds to the shortest repeated suffix that is always followed by the same letter or it is the longest context if such a suffix does not exist.

Question. Design a data structure able to maintain online the number of occurrences of each context and that can be managed in linear time on a constant-size alphabet.

Solution

The solution is based on a Prefix tree. The Prefix tree for $y[0\,..\,i]$ is constructed from the Prefix tree of $y[0\,..\,i-1]$ and essentially consists of the Suffix tree of $y[0\,..\,i]^{\mathrm{R}}$.

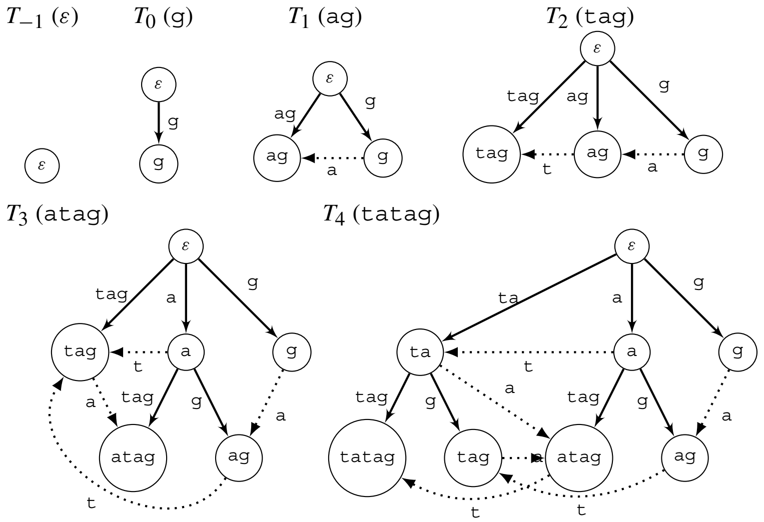
Let T_i denote the Prefix tree of $y[0\,..\,i]$. Its nodes are factors of $y[0\,..\,i]$. The initial tree T_{-1} is defined to be a single node. Prefix links are defined for every node of T_i except for its root and its most recent leaf. A prefix link labelled by letter a from node w points to node wa or to node uwa if every occurrences of wa are preceded by u.

Assume that the Prefix tree for $y[0\,..\,i-1]$ is build. Let *head*(w) denote the longest suffix of w that has an internal occurrence in w.

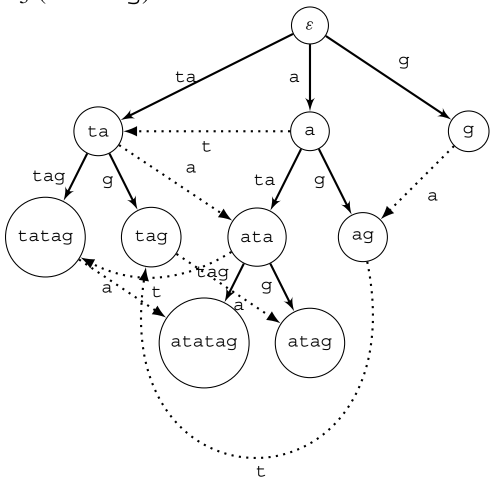
The Prefix tree is updated as follows. The insertion of $y[i]$ starts at the head of $w = y[0\,..\,i-1]$ and ends at the head of $w' = y[0\,..\,i]$. If $y[i]$ already occurred after w then the node w has a prefix link labelled by $y[i]$ that points to the head of w'. If w does not have a prefix link labelled by $y[i]$, the search proceeds with the parent of w until either a prefix link labelled by $y[i]$ is found or the root of the tree is reached. If the reached node p is w' then only a new leaf q is added to the tree. If the reached node p is uw' for some $u \in A^+$ then a new internal node r and a new leaf q are added to the tree.

All the nodes visited during the process now have a prefix link labelled by $y[i]$ pointing to the new leaf q. When a new internal node r is created some prefix links pointing to p may need to be updated to point to r.

Example. The pictures show the transformation of the Prefix tree when processing $y =$ `gatata`.



T_5 (atatag)



***Theorem* 15** *The above procedure correctly computes the Prefix tree T_i from the Prefix tree T_{i-1}.*

Proof T_{i-1} contains paths labelled by all the prefixes of $w = y[0\,..\,i-1]$ and only these paths. It then only misses a path labelled by $w' = y[0\,..\,i]$. Starting from the leaf s corresponding to w in T_{i-1} and going up to find the first node having a prefix link labelled by $a = y[i]$ identifies the node t corresponding to the longest suffix v of w such that va is a factor of w.

- If the prefix link from t labelled by a points to a node p corresponding to va then a new leaf q corresponding to w' must be added to the tree and the branch from p to q is labelled by u, where $w' = uva$. All the nodes scanned from s to t (except t itself) must now have a prefix link labelled by a pointing to q.
- If the prefix link from t labelled by a points to a node p corresponding to $v'va$ then a new internal node r corresponding to va is created having two successors: p and a new leaf q corresponding to w'. The branch from r to p must be labelled by v' and the branch from r to q must be labelled by u, where $w' = uva$. All the nodes scanned from s to t (except t itself) must now have a prefix link labelled by a pointing to q. Then prefix links going from nodes v', v' suffix of v, to p should now point to the new internal node r.

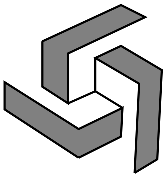
In both cases the tree now contains all the path contained in T_{i-1} and a path corresponding to w'. It is thus T_i. ■

***Theorem* 16** *The construction of T_{n-1} can be done in $O(n)$ time.*

Proof The running time of the construction is dominated by the number of nodes visited during each phase when computing $head(y[0\mathinner{..}i])$ for $0 \le i \le n-1$. Let k_i denote the number of nodes visited while searching for $head(y[0\mathinner{..}i])$. We have $|head(y[0\mathinner{..}i])| \le |head(y[0\mathinner{..}i-1])y[i]| - k_i$. Finally $\sum_0^{n-1} k_i = n - |head(y)| \le n$. Thus at most n nodes are visited during the whole construction of T_{n-1}, which proves the statement. ■

Notes

Prediction by Partial Matching was designed by Cleary and Witten [56] (see also [190]). PPM* was first introduced in [55]. The present Prefix tree construction is by Effros [107].



106 Compressing Suffix Arrays

Suffix arrays constitute a simple and space-economical data structure for indexing texts. In addition, there are many compressed versions of suffix arrays. The problem discusses one of them for compressing the array that stores the sorted (partial) list of suffixes (more precisely the partial rank array) of the concerned text. This shows an application of simple number theory.

Number-theoretic tools. A set $D \subseteq [0\mathinner{..}t-1]$ is called a *t-difference-cover* if all elements of the interval are differences modulo t of elements in D:

$$[0\mathinner{..}t-1] = \{(x-y) \bmod t : x, y \in D\}.$$

For example, the set $D = \{2,3,5\}$ is a 6-difference-cover for the interval $[0\mathinner{..}6]$ since $1 = 3-2$, $2 = 5-3$, $3 = 5-2$, $4 = 3-5 \pmod 6$ and $5 = 2-3 \pmod 6$.

It is known that for every positive integer t there is a t-difference-cover of size $O(\sqrt{t})$ and that the set can be constructed in time $O(\sqrt{t})$.

A set $\mathcal{S}(t) \subseteq [1\,..\,n]$ is called a *t-cover* of the interval $[1\,..\,n]$ if both $|\mathcal{S}(t)| = O\big(\frac{n}{\sqrt{t}}\big)$ and there is a constant-time computable function

$$h : [1\,..\,n-t] \times [1\,..\,n-t] \to [0\,..\,t]$$

that satisfies

$$0 \le h(i,j) \le t \text{ and } i + h(i,j), j + h(i,j) \in \mathcal{S}(t).$$

A t-cover can be obtained from a t-difference-cover $\mathcal{D}$ (of the interval $[0\,..\,t-1]$) by setting $\mathcal{S}(t) = \{i \in [1\,..\,n] : i \bmod t \in \mathcal{D}\}$. The following fact is known.

Fact. For each $t \le n$ a t-cover $\mathcal{S}(t)$ can be constructed in time $O(\frac{n}{\sqrt{t}})$.

Question. Show that the sorted partial list of suffixes of a text of length n can be represented in only $O(n^{3/4})$ amount of memory space and can still allow comparison of any two suffixes in $O(\sqrt{n})$ time.

[**Hint:** Use the notion of t-covers on intervals of integers.]

Solution

The answer to the question relies on t-covers. Instead of the array SA that stores the sorted list of suffixes of the text w, we use equivalently the array Rank, inverse of SA, that gives the ranks of suffixes indexed by their starting positions. With the whole array, comparing two suffixes starting at positions i and j amounts to comparing their ranks and takes constant time. However, the goal here is to retain only a small part of the table Rank.

Let $\mathcal{S}$ denote a fixed $\sqrt{n}$-cover $\{i_1, i_2, \ldots, i_k\}$ of $[1\,..\,n]$, where integers are sorted: $i_1 < i_2 < \cdots < i_k$. Its size is then $O(n^{3/4})$. Let $\mathcal{L}$ be the list of pairs

$$((i_1, \text{Rank}[i_1]), (i_2, \text{Rank}[i_2]), \ldots, (i_k, \text{Rank}[i_k])).$$

Since the list is sorted with respect to the first component of pairs, checking if a position i belongs to $\mathcal{S}$ and finding its rank in $\mathcal{L}$ can be done in logarithmic time.

Assume we want to compare lexicographically suffixes starting at positions i and j on w of length n. Let $\Delta = h(i,j)$.

The words $x[i\,..\,i+\Delta-1]$ and $x[j\,..\,j+\Delta-1]$ are first compared in a naive way (letter by letter), which takes $O(\Delta)$ time. If they match it remains to compare the suffixes starting at positions $i+\Delta$ and $j+\Delta$. The latter comparison takes logarithmic time because positions $i+\Delta$ and $j+\Delta$ are in $\mathcal{S}$ and we can recover their associated ranks from the list $\mathcal{L}$ in logarithmic time.

Altogether the comparison spends $O(\sqrt{n})$ time since $\Delta = O(\sqrt{n})$.

Example. The set $\mathcal{S}(6) = \{2,3,5,8,9,11,14,15,17,20,21,23\}$ is a 6-cover of $[1\,..\,23]$ built from the 6-difference-cover $\mathcal{D} = \{2,3,5\}$. In particular, we have $h(3,10) = 5$, since $3+5,\ 10+5 \in \mathcal{S}(6)$.

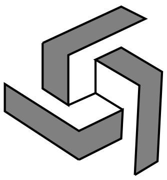
If we are to compare suffixes starting at positions 3 and 10 on the word w we have only to compare their prefixes of length 5 and possibly check whether $Rank[3+5] < Rank[10+5]$ or not.

Notes

By choosing $t = n^{2/3}$ instead of $\sqrt{n}$ in the proof the data structure is reduced to $O(t)$ memory space but then the time to compare two suffixes increases to $O(t)$.

The construction of difference-covers can be found in [178]. It is used to construct t-covers as done, for example, in [47], where the above fact is proved.

A similar method for compressed indexing is the notion of FM-index based on both Burrows–Wheeler transform and Suffix arrays. It has been designed by Ferragina and Manzini (see [112] and references therein). Its applications in Bioinformatics are described in the book by Ohlebusch [196].



107 Compression Ratio of Greedy Superstrings

The problem considers Algorithm GreedySCS (presented under different forms in Problem 61) that computes a superstring $Greedy(X)$ for a set X of words of total length n. The superstring can be viewed as a compressed text representing all words in X and from this point of view it is interesting to quantify the gain of representing X by a superstring.

Let $GrCompr(X) = n - |Greedy(X)|$ denote the compression achieved by the greedy algorithm. Similarly define $OptCompr(X) = n - |OPT(X)|$ where OPT is an optimal (unknown) superstring for X.

The fraction $\frac{GrCompr(X)}{OptCompr(X)}$ is called the *compression ratio* of Algorithm GreedySCS.

Question. Show the compression ratio of Algorithm GreedySCS is at least 1/2.

[**Hint:** Consider the overlap graph of the input set.]

Solution

It is more convenient to deal with the iterative version of Algorithm GreedySCS from Problem 61.

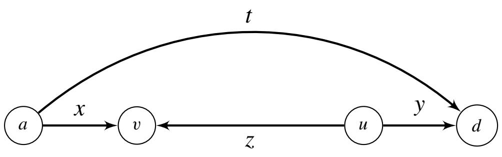
```
IterativeGreedySCS($X$ non-empty set of words)
1  while $|X| > 1$ do
2      let $x, y \in X, x \neq y$, with $|Overlap(x,y)|$ maximal
3      $X \leftarrow X \setminus \{x,y\} \cup \{x \otimes y\}$
4  return $x \in X$
```

Let us start with an abstract problem for directed graphs. Assume G is a complete directed graph whose edges are weighted by non-negative integers. If $u \to v$ is an edge the operation *contract*(u,v) identifies u, v and removes the edges out-going from u and in-going to v.

Let *OptHam*(G) be the maximum weight of a Hamiltonian path in G and *GreedyHam*(G) be the weight of the Hamiltonian path implicitly produced by the greedy algorithm. In each step the greedy algorithm for graphs chooses an edge $u \to v$ of maximum weight and applies *contract*(u,v). It stops when G becomes a single node. The chosen edges compose the resulting Hamiltonian path.

Relation between greedy superstring and greedy Hamiltonian path. We introduce the *overlap graph* G of a set X of words. The set of nodes is X and the weight of $x_i \to x_j$ is the maximal overlap between words x_i and x_j. Observe that the statement in line 3 of Algorithm IterativeGreedySCS corresponds to the operation *contract*(x,y). This implies the following fact.

Observation. The greedy Hamiltonian path for the overlap graph of X corresponds to the greedy superstring of X.



We say that a weighted graph G satisfies condition $(*)$ if in each configuration of the type shown on the above picture we have

$$z \geq x, y \Longrightarrow z + t \geq x + y.$$

Moreover, we require that it holds for each graph obtained from G by applying any number of contractions.

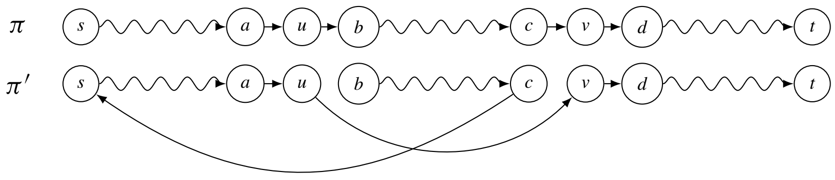
We leave the following technical but easy proof of the following fact about overlap graphs to the reader (see Notes).

***Lemma* 17** *The overlap graph satisfies condition* $(*)$.

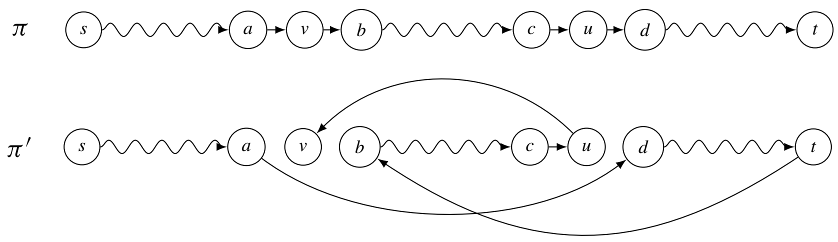
***Lemma* 18** *Assume G satisfies condition $(*)$, e is an edge of maximal weight z and G' is obtained from G by applying $contract(e)$. Then $OptHam(G') \geq OptHam(G) - 2z$.*

Proof Let $e = u \to v$. Let π be an optimal Hamiltonian path in G. Denote by $|\pi|$ the total weight of π. It is enough to show that any Hamiltonian path in G' is of weight at least $|\pi| - 2z$ or (equivalently) to show that any Hamiltonian path π' in G containing the edge $u \to v$ is of total weight at least $|\pi| - z$.

Case v is after u on π. We remove two edges (u, b) and (c, v) of weights at most z and insert edges (u, v) and (c, s) to get a new Hamiltonian path π' (see picture below). Contracting (u, v) decreases the total weight of the path by the sum of weights of (u, b) and (c, v), that is by at most $2z$. We get a Hamiltonian path in G' of total weight at least $|\pi| - 2z$; see the picture below. Consequently $OptHam(G') \geq OptHam(G) - 2z$.



Case v is before u on π. We use condition $(*)$ with $x = weight(a, v)$, $y = weight(u, d)$, $z = weight(u, v)$ and $t = weight(a, d)$. Let $q = weight(v, b)$. Let π' derived from π as in the picture below.



Due to condition $(*)$ and inequality $q \leq z$ we have

$$|\pi'| \geq |\pi| - x - y + z + t - q \geq |\pi| - z.$$

Consequently $|\pi'| \geq |\pi| - 2z$ and $OptHam(G') \geq OptHam(G) - 2\,z$. This completes the proof of Lemma 8. ■

***Lemma* 19** *If G satisfies condition* $(*)$ *then* $GreedyHam(G) \geq \frac{1}{2}OptHam(G)$.

Proof The proof is by induction on the number of nodes of G. Let z be the maximum weight of an edge in G whose contraction gives G'.

On the one hand, G' is a graph smaller than G, so applying the inductive assumption we have $GreedyHam(G') \geq \frac{1}{2}OptHam(G')$. On the other hand, we have $OptHam(G') \geq OptHam(G') - 2z$ and $GreedyHam(G) = GreedyHam(G') + z$.
Thus: $GreedyHam(G) \geq \frac{1}{2}OptHam(G') + z \geq \frac{1}{2}(OptHam(G) - 2z) + z$ $\geq \frac{1}{2}OptHam(G)$, which proves the statement. ■

The above observation and Lemmas 17, 18 and 19 imply directly that the greedy algorithm for superstrings achieves a 1/2 compression ratio.

Notes

The present proof of the problem is a version of the proof given by Tarhio and Ukkonen in [230].