Space-Efficient Re-Pair Compression

Philip Bille*, Inge Li Gørtz*, and Nicola Prezza†

*Technical University of Denmark
2800 Kgs. Lyngby
Copenhagen, Denmark
{phbi,inge}@dtu.dk

†University of Udine
Via delle scienze, 206
33100, Udine, Italy
prezza.nicola@spes.uniud.it

Abstract

Re-Pair [1] is an effective grammar-based compression scheme achieving strong compression rates in practice. Let n, σ, and d be the text length, alphabet size, and dictionary size of the final grammar, respectively. In their original paper, the authors show how to compute the Re-Pair grammar in expected linear time and $5n + 4\sigma^2 + 4d + \sqrt{n}$ words of working space on top of the text. In this work, we propose two algorithms improving on the space of their original solution. Our model assumes a memory word of $\lceil \log_2 n \rceil$ bits and a re-writable input text composed by n such words. Our first algorithm runs in expected $\mathcal{O}(n/\epsilon)$ time and uses $(1+\epsilon)n+\sqrt{n}$ words of space on top of the text for any parameter $0 < \epsilon \leq 1$ chosen in advance. Our second algorithm runs in expected $\mathcal{O}(n \log n)$ time and improves the space to $n + \sqrt{n}$ words.

1 Introduction

Re-Pair (short for recursive pairing) is a grammar-based compression invented in 1999 by Larsson and Moffat [1]. Re-Pair works by replacing a most frequent pair of symbols in the input string by a new symbol, reevaluating the all new frequencies on the resulting string, and then repeating the process until no pairs occur more than once. Specifically, on a string S, Re-Pair works as follows. (1) It identifies the most frequent pair of adjacent symbols ab. If all pair occur once, the algorithm stops. (2) It adds the rule $A \rightarrow ab$ to the dictionary, where A is a new symbol not appearing in S. (3) It repeats the process from step (1).

Re-Pair achieves strong compression ratios in practice and in theory [2–5]. Re-Pair has been used in wide range of applications, e.g., graph representation [4], data mining [?], and tree compression [?].

Let n, σ, and d denote the text length, the size of the alphabet, and the size of the dictionary grammar, respectively. Larsson et al. [1] showed how to implement Re-Pair in $O(n)$ expected time and $5n + 4\sigma^2 + 4d + \sqrt{n}$ words of space in addition to the text. [1] The space overhead is due to several data structures used to track the pairs to be replaced and their frequencies. As noted by several authors this makes Re-Pair problematic to apply on large data, and various workarounds have been devised (see e.g. [2–4]).

[1] For simplicity, we ignore any additive $+O(1)$ terms in all space bounds.

Surprisingly, the above bound of the original paper remains the best known complexity for computing the Re-Pair compression. In this work, we propose two algorithms that significantly improve this bound. As in the previous work we assume a standard unit cost RAM with memory words of $\lceil \log_2 n \rceil$ bits and that the input string is given in n such word. Furthermore, we assume that the input string is *re-writeable*, that is, the algorithm is allowed modify the input string during execution, and we only count the space used in addition this string in our bounds. Since Re-Pair is defined by a repeated re-writing operations, we believe this is natural model for studying this type of compression scheme. Note that we can trivially convert any algorithm with a re-writeable input string to a read-only input string by simply copying the input string to working memory, at the cost of only n extra words of space.

Theorem 1. *Given a re-writeable string S of length n we can compute the Re-Pair compression of S in*

(i) $\mathcal{O}(n/\epsilon)$ expected time and $(1+\epsilon)n + \sqrt{n}$ words of space for any $0 < \epsilon \leq 1$, or

(ii) $\mathcal{O}(n \log n)$ expected time and $n + \sqrt{n}$ words.

Note that since $\epsilon = O(1)$ the time in Thm. 1(i) is always at least $\Omega(n)$. For any constant ϵ, (i) matches the optimal linear time bound of Larsson and Moffat [1], while improving the leading space term by almost $4n$ words to $(1+\epsilon)n + \sqrt(n)$ words (with careful implementation it appears that [1] may be implemented to exploit a re-writeable input string. If so, our improvement is instead almost $3n$ words.) Thm. 1(ii) further improves the space to $n + \sqrt{n}$ at the cost of increasing time by a logarithmic factor. By choosing $\epsilon = o(1/\log n)$ the time in (i) is faster than (ii) at the cost of a slight increase in space. For instance, with $\epsilon = 1/\log\log n$ we obtain $\mathcal{O}(n \log\log n)$ time and $n + \sqrt{n} + \log\log n$ words.

Our algorithm consists of two main phases: high-frequency and low-frequency pair processing. We define a *high-frequency* (resp. *low frequency*) pair a character pair appearing at least (resp. less than) $\lceil \sqrt{n}/3 \rceil$ times in the text (we will clarify later the reason for using constant 3). Note that there cannot be more than $3\sqrt{n}$ distinct high-frequency pairs. Both phases use two data structures: a queue $\mathcal{Q}$ storing character pairs (prioritized by frequency) and an array TP storing text positions sorted by character pairs. $\mathcal{Q}$'s elements point to ranges in TP corresponding to all occurrences of a specific character pair. In Section 4.2 we show how we can sort in-place and in linear time any subset of text positions by character pairs. The two phases work exactly in the same way, but use two different implementations for the queue giving different space/time tradeoffs for operations on it. In both phases, we extract (high-frequency/low-frequency) pairs from $\mathcal{Q}$ (from the most to least frequent) and replace them in the text with fresh new dictionary symbols.

When performing a pair replacement $A \to ab$, for each text occurrence of ab we replace a with A and b with the blank character '␣'. This strategy introduces a potential problem: after several replacements, there could be long (super-constant size) runs of blanks. This could increase the cost of reading pairs in the text by too much. In Section 4.1 we show how we can perform pair replacements while keeping the cost of skipping runs of blanks constant.

2 Preliminaries

Let n be the input text's length. Throughout the paper we assume a memory word of size $\lceil \log_2 n \rceil$ bits, and a rewritable input text T on an alphabet Σ composed by n such words. In this respect, the working space of our algorithms is defined as the amount of memory used *on top* of the input. Our goal is to minimize this quantity while achieving low running times. For reasons explained later, we reserve two characters (*blank* symbols) denoted as '*' and '_'. We encode these characters with the integers $n-2$ and $n-1$, respectively [2].

The Re-Pair compression scheme works by replacing character pairs (with frequency at least 2) with fresh new symbols. We use the notation $\mathcal{D}$ to indicate the *dictionary* of such new symbols, and denote by $\bar{\Sigma}$ the extended alphabet $\bar{\Sigma} = \Sigma \cup \mathcal{D}$. It is easy to prove (by induction) that $|\bar{\Sigma}| \leq n$: it follows that we can fit both alphabet characters and dictionary symbols in $\lceil \log_2 n \rceil$ bits. The output of our algorithms consists in a set of rules of the form $X \rightarrow AB$, with $A, B \in \bar{\Sigma}$ and $X \in \mathcal{D}$. Our algorithms stream the set of rules directly to the output (e.g. disk), so we do not count the space to store them in main memory.

3 Main Algorithm

We describe our strategy top-down: first, we introduce the queue $\mathcal{Q}$ as a blackbox, and use it to describe our main algorithm. In the next sections we describe the high-frequency and low-frequency pair processing queues implementations.

3.1 The queue as a blackbox

Our queues support the following operations:

- $\mathbf{new\_low\_freq\_queue(T, TP)}$. Return the low-frequency pairs queue
- $\mathbf{new\_high\_freq\_queue(T, TP)}$. Return the high-frequency pairs queue
- $\mathcal{Q}[\mathbf{ab}]$, $ab \in \bar{\Sigma}^2$. If ab is in the queue, return a triple $\langle P_{ab}, L_{ab}, F_{ab} \rangle$, with $L_{ab} \geq F_{ab}$ such that:

[2] If the alphabet size is $|\Sigma| < n-1$, then we can reserve the codes $n-2$ and $n-1$ without increasing the number of bits required to write alphabet characters. Otherwise, if $|\Sigma| \geq n-1$ note that the two (or one) alphabet characters with codes $n-2 \leq x, y \leq n-1$ appear in at most two text positions i_1 and i_2, let's say $T[i_1] = x$ and $T[i_2] = y$. Then, we can overwrite $T[i_1]$ and $T[i_2]$ with the value 0 and store separately two pairs $\langle i_1, x \rangle$, $\langle i_2, y \rangle$. Every time we read a value $T[j]$ equal to 0, in constant time we can discover whether $T[j]$ contains 0, x, or y. Throughout the paper we will therefore assume that $|\Sigma| \leq n$ and that characters from $\Sigma \cup \{*, \_\}$ fit in $\lceil \log_2 n \rceil$ bits.

(i) ab has frequency F_{ab} in the text

(ii) All text occurrences of ab are contained in $TP[P_{ab}, \ldots, P_{ab} + L_{ab} - 1]$

- $\mathcal{Q}$.**max**()/$\mathcal{Q}$.**min**(): return the pair ab in $\mathcal{Q}$ with the highest/lowest F_{ab}
- $\mathcal{Q}$.**remove**(**ab**): delete ab from $\mathcal{Q}$
- $\mathcal{Q}$.**contains**(**ab**): return true iff $\mathcal{Q}$ contains pair ab
- $\mathcal{Q}$.**size**() return the number of pairs stored in $\mathcal{Q}$
- $\mathcal{Q}$.**decrease**(**ab**): decrease F_{ab} by one
- $\mathcal{Q}$.**synchronize**(**AB**). If $F_{AB} < L_{AB}$, then $TP[P_{AB}, \ldots, P_{AB} + L_{AB} - 1]$ contains occurrences of pairs $XY \neq AB$ (and/or blank positions). The procedure sorts $TP[P_{AB}, \ldots, P_{AB}+L_{AB}-1]$ by character pairs (ignoring positions containing a blank) and, for each such XY, removes the least frequent pair in $\mathcal{Q}$ and creates a new queue element for XY pointing to the range in TP corresponding to the occurrences of XY. If XY is less frequent than the least frequent pair in $\mathcal{Q}$, XY is not inserted in the queue. Before exiting, the procedure re-computes P_{AB} and L_{AB} so that $TP[P_{AB}, \ldots, P_{AB} + L_{AB} - 1]$ contains all and only the occurrences of AB in the text (in particular, $L_{AB} = F_{AB}$)

3.2 Algorithm

In Algorithm 1 we describe the procedure substituting the most frequent pair in the text with a fresh new dictionary symbol. We use this procedure in Algorithm 2 to compute the re-pair grammar. Variables T (the text), TP (array of text positions), and X (next free dictionary symbol) are global, so we do not pass them from Algorithm 2 to Algorithm 1. Note that—in Algorithm 1—new pairs appearing after a substitution can be inserted in $\mathcal{Q}$ only inside procedure $\mathcal{Q}$.*synchronize* at Lines 14, and 15. However, operation at Line 14 is executed only under a certain condition. As discussed in the next sections, this trick allows us to amortize operations while preserving correctness of the algorithm.

In Lines 4, 5, and 12 of Algorithm 1 we assume that—if necessary—we are skipping runs of blanks while extracting text characters (constant time, see Section 4.1). In Line 5 we extract AB and the two symbols x, y preceding and following it (skipping runs of blanks if necessary). In Line 12, we extract a text substring s composed by X and the symbol preceding it (skipping runs of blanks if necessary). After this, we replace each X with AB in s and truncate s to its suffix of length 3.

3.3 Amortization: correctness and complexity

Assuming the correctness of the queue implementations (see next sections), all we are left to show is the correctness of our amortization policy at Lines 13 and 14 of Algorithm 1. More formally: in Algorithm 1, replacements create new pairs; however, to amortize operations we postpone the insertion of such pairs in the queue (Line 14 of Algorithm 1). To prove the correctness of our algorithm, we need to show that

```
Algorithm 1: $substitution\_round(\mathcal{Q})$
input    : The queue $\mathcal{Q}$
behavior: Pick the most frequent pair from $\mathcal{Q}$ and replace its occurrences in
          the text with a new dictionary symbol
1  $AB \leftarrow \mathcal{Q}.max()$;
2  $\overline{ab} \leftarrow \mathcal{Q}.min()$;      /* global variable storing least frequent pair */
3  output $X \rightarrow AB$;                                  /* output new rule */
4  for $i = TP[P_{AB}], \dots, TP[P_{AB} + L_{AB} - 1]$ and $T[i, i+1] = AB$ do
5      $xABy \leftarrow get\_context(T, i)$; /* $AB$'s context (before replacement) */
6      $replace(T, i, X)$;          /* Replace $X \rightarrow AB$ at position $i$ in $T$ */
7      if $\mathcal{Q}.contains(xA)$ then
8          $\mathcal{Q}.decrease(xA)$;
9      if $\mathcal{Q}.contains(By)$ then
10         $\mathcal{Q}.decrease(By)$;
11 for $i = TP[P_{AB}], \dots, TP[P_{AB} + L_{AB} - 1]$ and $T[i] = X$ do
12     $xAB \leftarrow get\_context'(T, i)$;                 /* $X$'s left context */
13     if $\mathcal{Q}.contains(xA)$ and $F_{xA} \leq L_{xA}/2$ then
14         $\mathcal{Q}.synchronize(xA)$;
15 $\mathcal{Q}.synchronize(AB)$;   /* Find new pairs in $AB$'s occurrences list */
16 $\mathcal{Q}.remove(AB)$;
17 $X \leftarrow X + 1$;                               /* New dictionary symbol */
```

every time we pick the maximum AB from $\mathcal{Q}$ (Line 1, Algorithm 1), AB is the pair with the highest frequency in the text (i.e. all postponed pairs have lower frequency than AB). Suppose, by contradiction, that at Line 1 of Algorithm 1 we pick pair AB, but the highest-frequency pair in the text is $CD \neq AB$. Since CD is not in $\mathcal{Q}$, we have that (i) CD appeared after some substitution $D \rightarrow zw$ which generated occurrences of CD in portions of the text containing Czw, and[3] (ii) $F_{Cz} > L_{Cz}/2$, otherwise the synchronization step at Line 14 of Algorithm 1 ($\mathcal{Q}.synchronize(Cz)$) would have been executed, and CD would have been inserted in $\mathcal{Q}$. Note that all occurrences of CD are contained in $TP[P_{Cz}, \dots, P_{Cz} + L_{Cz} - 1]$. $F_{Cz} > L_{Cz}/2$ means that *more than half* of the entries $TP[P_{Cz}, ..., P_{Cz} + L_{Cz} - 1]$ contain an occurrence

[3] Note that, if CD appears after some substitution $C \rightarrow zw$ which creates occurrences of CD in portions of the text containing zwD, then all occurrences of CD are contained in $TP[P_{zw}, \dots, P_{zw} + L_{zw} - 1]$, and we insert CD in $\mathcal{Q}$ at Line 15 of Algorithm 1 within procedure $\mathcal{Q}.synchronize(zw)$

Algorithm 2: *compute_repair*(T)

```
input    : Text $T \in \Sigma^n$
behavior: The re-pair grammar of $T$ is computed and streamed to output

$n \leftarrow |T|$;
$X \leftarrow |\Sigma|$;                              /* next dictionary symbol */
while $highest\_frequency(T) \geq \sqrt{n}/3$ do
    $TP \leftarrow sort\_pairs(T)$;                /* sort $T$'s positions by pairs */
    $\mathcal{Q} \leftarrow new\_high\_freq\_queue(T, TP)$;
    while $\mathcal{Q}.size() > 0$ do
        $substitution\_round(\mathcal{Q})$;
    $free\_memory(\mathcal{Q}, TP)$;
    $compact\_text(T)$;                                        /* delete blanks */
while $highest\_frequency(T) > 2$ do
    $TP \leftarrow sort\_pairs(T)$;                /* sort $T$'s positions by pairs */
    $\mathcal{Q} \leftarrow new\_low\_freq\_queue(T, TP)$;
    while $\mathcal{Q}.size() > 0$ do
        $substitution\_round(\mathcal{Q})$;
    $free\_memory(\mathcal{Q}, TP)$;
    $compact\_text(T)$;                                        /* delete blanks */
```

of Cz, which implies than *less than half* of such entries contain occurrences of pairs different than Cz (in particular CD, since $D \neq z$). This, combined with the fact that all occurrences of CD are stored in $TP[P_{Cz}, ..., P_{Cz} + L_{Cz} - 1]$, yields $F_{CD} \leq L_{Cz}/2$. Then, $F_{CD} \leq L_{Cz}/2 < F_{Cz}$ means that Cz has a higher frequency than CD. This leads to a contradiction, since we assumed that CD was the pair with the highest frequency in the text.

Note that operation $\mathcal{Q}.synchronize(xA)$ at Line 14 of Algorithm 1 scans xA's occurrences list ($\Theta(L_{xA})$ time). However, to keep time under control, in Algorithm 1 we are allowed to spend only time proportional to F_{AB}. Since L_{xA} could be much bigger than F_{AB}, we need to show that our strategy amortizes operations. Consider an occurrence $xABy$ of AB in the text. After replacement $X \rightarrow AB$, this text substring becomes xXy. In Lines 8-10 we decrease by one in constant time the two frequencies F_{xA} and F_{By} (if they are stored in $\mathcal{Q}$). Note: we manipulate just F_{xA} and F_{By}, and not the actual intervals associated with these two pairs. As a consequence, for a general pair ab in $\mathcal{Q}$, values F_{ab} and L_{ab} do not always coincide. However, we make sure that, when calling $\mathcal{Q}.max()$ at Line 1 of Algorithm 1, the following invariant holds for every pair ab in the priority queue:

$$F_{ab} > L_{ab}/2$$

The invariant is maintained by calling $\mathcal{Q}.synchronize(xA)$ (Line 14, Algorithm 1) as soon as we decrease by “too much” F_{xA} (i.e. $F_{xA} \leq L_{xA}/2$). It is easy to see that this policy amortizes operations: every time we call procedure $\mathcal{Q}.synchronize(ab)$, either—Line 15—we are replacing ab with a fresh new dictionary symbol (thus $L_{ab} < 2 \cdot F_{ab}$ work is allowed), or—Line 14—we just decreased F_{ab} by too much ($F_{ab} \leq L_{ab}/2$). In the latter case, we already have done at least $L_{ab}/2$ work during previous replacements (each one has decreased ab's frequency by 1), so $\mathcal{O}(L_{ab})$ additional work does not asymptotically increase running times.

4 Details and Analysis

We first describe how we implement character replacement in the text and how we efficiently sort text positions by pairs. Then, we provide the two queue implementations. For the low-frequency pairs queue, we provide two alternative implementations leading to two different space/time tradeoffs for our main algorithm. We conclude by analyzing the complexity of Algorithm 2 with the different queue implementations.

4.1 Skipping blanks in constant time

As noted above, pair replacements generate runs of the blank character ’_’. Our aim in this section is to show how to skip these runs in constant time. Recall that the text is composed by $\lceil \log_2 n \rceil$-bits words. Recall that we reserve *two* blank characters: ’*’ and ’_’. If the run length r satisfies $r < 10$, then we fill all run positions with character ’_’ (skipping this run takes constant time). Otherwise, ($r \geq 10$) we start and end the run with the string `_*i*_`, where $i = r - 1$, and fill the remaining run positions with ’_’. For example, the text `aB___________c` is stored as

`a B _ * 10 * _ _ _ * 10 * _ c`

Note that, with this solution, only run lengths are delimited by character ’*’: it follows that we can distinguish between run lengths and alphabet characters. We remind the reader that we encode the extra characters ’*’ and ’_’ with the integers $n - 2$ and $n - 1$, respectively. Then, it is easy to see that any integer i storing a run length (minus 1) always satisfies $i \leq n - 3$, so we can safely distinguish between run lengths and reserved blank characters. Our text representation is completely transparent in that it allows to retrieve any text character/blank in constant time: if $T[j] > n-3$, then $T[j]$ contains a blank. If $T[j] \leq n-3$, if $T[j-1] = T[j+1] = n-2$ then $T[j]$ contains a blank, otherwise an alphabet character.

The only thing we are left to show is how to merge two runs of blanks in the case a single alphabet character between them is replaced by a blank after a substitution. For example, the text

```
a B _ * 10 * _ _ _ * 10 * _ C _ * 11 * _ _ _ _ * 11 * _ D
```

after substitution $E \rightarrow BC$ should become

```
a E _ * 23 * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * 23 * _ D
```

It is easy to see that the replacement can be implemented in constant time starting from the position containing 'B', so for space reasons we do not discuss it further. In the paper we assume that the text is stored with the above representation, with constant-time cost for skipping runs of blanks.

4.2 Sorting pairs and frequency counting

Let T be an array of n entries each consisting of a word of $\log n$ bits (for simplicity, we assume that n is a power of two). In this section we show how to sort the pairs in T lexicographically in linear time using additional n words of memory. Our algorithm only requires read-only access to T. Furthermore, the algorithm generalizes substrings of any constant length in the same complexity. As an immediate corollary, we can compute the frequency of each pair in the same complexity simply by traversing the sorted sequence.

Our solution needs the following results on in-place sorting and merging.

Lemma 1 (Franceschini et al. [6])**.** *Given an array A of length n with $\mathcal{O}(\log n)$ bit entries, we can in-place sort A in $\mathcal{O}(n)$ time.*

Lemma 2 (Salowe and Steiger [7])**.** *Given arrays A and B of total length n, we can merge A and B in-place using a comparison-based algorithm in $\mathcal{O}(n)$ time.*

The above result immediately provides simple but inefficient solutions to sorting pairs. In particular, we can copy each pair of T into an array of n entries each storing a pair using 2 words, and then in-place sort the array using Lemma 1. This uses $\mathcal{O}(n)$ time but requires $2n$ words space. Alternatively, we can copy the positions of each pair into an array and then apply a comparison-based in-place sorting algorithm based on 2. This uses $\mathcal{O}(n \log n)$ time but only requires n words of space. Our result simultaneously obtains the best of these time and space bounds.

Our algorithm works as follows. Let A be an array of n words. We greedily process T from left-to-right in phases. In each phase we process a contiguous segment $T[i, j]$ of overlapping pairs of T and compute and store the corresponding sorted segment in $A[i, j]$. Phase $i = 0, \ldots, k$ proceeds as follows. Let r_i denote the number of *remaining pairs* in T not yet processed. Initially, we have that $r_0 = n$. Note that r_i is also the number of unused entries in A. We copy the next $r_i/3$ pairs of T into A. Each pair is encoded using the two characters of the pair and the position of the pair in T. Hence, each encoded pair uses 3 words and thus fills all remaining r_i entries in A. We sort the encoded segment using the in-place sort from Lemma 1, where each 3-words encoded pair is viewed as a single key. We then compact the segment back into $r_i/3$ only entries of A by throwing away the characters of each pair and only keeping the position of the pair. We repeat the process until all pairs in T have been processed. At the end A consists of a collection of segments of sorted pairs. We merge the segments from right-to-left using the in-place comparison-based merge from Lemma 2 (note that segment borders can be detected by accessing the text, so we do not need to store them separately).

Next we analyse the algorithm. For the space bound, note that at any point in time the algorithm never uses more than $\mathcal{O}(1)$ words in addition to the remaining entries in A. Hence, the total space is $n + \mathcal{O}(1)$ words. For the time bound, note that each phase decreases the number of remaining pairs in A by a third of the current number of remaining pairs. Hence, for $i > 0$, $r_i = r_{i-1} - r_{i-1}/3 = (2/3)r_i$. Since $r_0 = n$ it follows that $r_i = (2/3)^i n$. The total number of phases is thus $k = \log_{3/2} n$. By Lemma 1 phase i uses $\mathcal{O}(r_i)$ time and hence the total time for all phases is $\sum_i^k \mathcal{O}(r_i) = \mathcal{O}(n)$. For the merging step, note that the last step merges two segments of total size n, the second last step merges two segments of total size $(2/3)n$, and in general the ith last step merges two segments of total size $(2/3)^i n$. By Lemma 2 the total time is thus $\sum_i^k \mathcal{O}((2/3)^i n) = \mathcal{O}(n)$. In summary, we have the following result for sorting and immediately corollary for counting frequencies.

Lemma 3. *Given a string T of length n with $\lceil \log_2 n \rceil$-bit characters, we can sort the pairs of T in $\mathcal{O}(n)$ time using $n + \mathcal{O}(1)$ words.*

Lemma 4. *Given a string T of length n with $\lceil \log_2 n \rceil$-bit characters, we can count the frequencies of pairs of T in $\mathcal{O}(n)$ time using $n + \mathcal{O}(1)$ words.*

We use Lemma 4 to find the maximum frequency in linear time and n words of space in Algorithm 2 (procedure $highest\_frequency(T)$). Note that our technique can be used to sort in-place and linear time any subset of text positions by character pairs (required in our queue implementations). Finally, note that with our strategy text positions are sorted first by character pairs, and then by position (this follows from the fact that—inside our sorting procedure—we concatenate the pair and the text position in a single integer of 3 words). This fact is important in our algorithm since it allows us to process text pairs from left to right.

4.3 High-Frequency Pairs Queue

The capacity of the high-frequency pairs queue is $\sqrt{n}/11$. We implement $\mathcal{Q}$ with the following two components:

(i) **Hash** $\mathcal{H}$. We keep a hash table $\mathcal{H} : \bar{\Sigma}^2 \rightarrow [0, \sqrt{n}/11]$ with $(2/11)\sqrt{n}$ entries.

$\mathcal{H}$ will be filled with at most $\sqrt{n}/11$ pairs (hash load ≤ 0.5). Collisions are solved by linear probing. The overall size of the hash is $(6/11)\sqrt{n}$ words: 3 words (one pair and one integer) per hash entry.
(ii) **Queue array** B. We keep an array B of quadruples from $\bar{\Sigma}^2 \times [0,n) \times [0,n) \times [0,n)$. B will be filled with at most $\sqrt{n}/11$ entries. We denote with $\langle ab, P_{ab}, L_{ab}, F_{ab} \rangle$ a generic element of B. The idea is that B stores the most frequent character pairs, together with their frequencies. Every time we pop the highest-frequency pair from the queue, the following holds: (i) ab has frequency F_{ab} in the text, and (ii) ab occurs in a *subset* of text positions $TP[P_{ab}], \ldots, TP[P_{ab} + L_{ab} - 1]$. The overall size of B is $(5/11)\sqrt{n}$ words.

$\mathcal{H}$'s entries point to B's entries: at any stage of the algorithm, if $\mathcal{H}$ contains a pair ab, then $B[\mathcal{H}[ab]] = \langle ab, P_{ab}, L_{ab}, F_{ab} \rangle$ is the quadruple associated with the pair. Overall, $\mathcal{Q} = \langle \mathcal{H}, B \rangle$ takes $\sqrt{n}$ words of space. Figure 1 depicts our high-frequency queue.

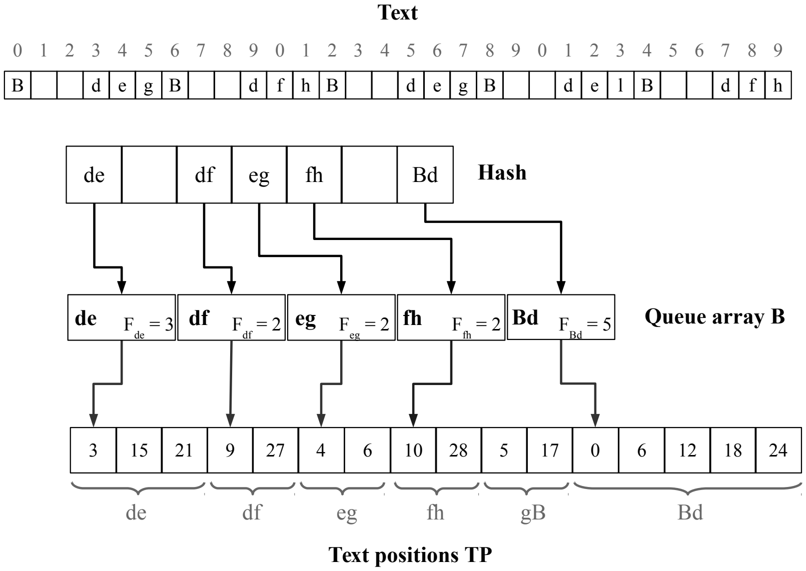

Figure 1: The high-frequency queue (hash and queue array). In this example, the text is obtained from `abcdegabcdfhabcdegabcdelabcdfh` after the substitutions (in order) $A \rightarrow ab$ and $B \rightarrow Ac$. Pairs de, df, eg, fh, and Bd have been inserted in the queue. In this example, the L-field of B elements is always equal to the F-field, so we do not show it to improve readability.

First, we show how function $new\_high\_freq\_queue(T, TP)$ is implemented. We scan TP from left to right and replace every maximal sub-array $TP[i, \ldots, i+k-1]$, $k \geq 2$, corresponding to a character pair $T[TP[i], TP[i]+1]$ with the integers pair $\langle k, TP[i] \rangle$. We store such pair in two words by concatenating the integers k and $TP[i]$.

Whenever $k = 1$ or $k > 2$, we compact TP positions so that all pairs are stored consecutively. In the end, (an opportune prefix of) TP contains a list $\langle k_1, j_1 \rangle, \dots, \langle k_t, j_t \rangle$ of pairs representing the frequency of all character pairs with frequency at least 2: $\langle k, j \rangle$ is in this list iff pair $T[j]T[j+1]$ appears $k \geq 2$ times in the text. We conclude by sorting this list in decreasing frequency order with the Algorithm described in [6]. Then, we scan this list left-to-right and insert at most $\sqrt{n}/11$ high-frequent pairs in $\mathcal{H}$, starting from the most frequent ones. Finally, we re-build TP (re-using the memory already allocated for it), and scan it left-to-right. For each maximal $TP[p, \dots, p+k-1]$ corresponding to a pair ab, if ab is in $\mathcal{H}$ then we set $\mathcal{H}[ab] \leftarrow |B|$ and append $\langle ab, p, k, k \rangle$ at the end of B. This procedure runs in $\mathcal{O}(n)$ time and uses $\mathcal{O}(1)$ words of space in addition to T, TP, B, and $\mathcal{H}$. Operations on $\mathcal{Q}$ are implemented as follows:

- $\mathcal{Q}[ab]$, $ab \in \bar{\Sigma}^2$: return the last three components of the quadruple $B[\mathcal{H}[ab]]$. $\mathcal{O}(1)$ expected time.
- $\mathcal{Q}.max()$ is implemented by scanning B. $\mathcal{O}(\sqrt{n})$ time.
- $\mathcal{Q}.min()$ is implemented by scanning B. $\mathcal{O}(\sqrt{n})$ time.
- $\mathcal{Q}.remove(ab)$: set $B[\mathcal{H}[ab]] \leftarrow \langle NULL, NULL, NULL, NULL \rangle$ and delete ab from $\mathcal{H}$. $\mathcal{O}(1)$ expected time.
- $\mathcal{Q}.contains(ab)$. This operation requires one access to $\mathcal{H}$. $\mathcal{O}(1)$ expected time.
- $\mathcal{Q}.size()$: we only need to keep a variable storing the current size and update it at each remove operation. $\mathcal{O}(1)$ time.
- $\mathcal{Q}.decrease(AB)$: decrease the fourth component of $B[\mathcal{H}[AB]]$ by one. If F_{AB} becomes equal to 1, remove the pair from the queue. $\mathcal{O}(1)$ expected time.
- $\mathcal{Q}.synchronize(AB)$. Let $\overline{ab}$ be a global variable storing the pair with the smallest frequency in $\mathcal{Q}$ (Algorithm 1, Line 2). We apply the sorting algorithm of Section 4.2 to sub-array $TP' = TP[P_{AB}, \dots, P_{AB} + L_{AB} - 1]$, excluding text positions that contain a blank character. After this, positions are clustered in TP' by character pairs. For each contiguous maximal sub-array $TP[p, \dots, p+k-1]$ of TP' corresponding to a pair XY:
  - If $XY \neq AB$ and $k > F_{\overline{ab}}$, we remove $\overline{ab}$ from $\mathcal{Q}$ and insert XY as follows. Let $j_{min} = \mathcal{H}[\overline{ab}]$. We remove $\overline{ab}$ from the hash $\mathcal{H}$, overwrite $B[j_{min}] \leftarrow \langle XY, p, k, k \rangle$, and insert $\mathcal{H}[XY] \leftarrow j_{min}$ in the hash. Then, we recompute $\overline{ab} = \mathcal{Q}.min()$.
  - If $XY = AB$, then we just overwrite $B[\mathcal{H}[AB]] \leftarrow \langle AB, p, k, k \rangle$ and set $\overline{ab} \leftarrow AB$ if $k < F_{\overline{ab}}$.

Running time: $\mathcal{O}(L_{AB} + N \cdot \sqrt{n})$, where L_{AB} is AB's interval length at the moment of entering in this procedure, and N is the number of new pairs XY inserted in the queue (we call $\mathcal{Q}.min()$ for each of them).

Time complexity

Note that to find the most frequent pair in $\mathcal{Q}$ we scan all $\mathcal{Q}$'s elements; since $|\mathcal{Q}| \in \mathcal{O}(\sqrt{n})$ and there are at most $3\sqrt{n}$ high-frequency pairs, the overall time spent inside procedure $max(\mathcal{Q})$ does not exceed $\mathcal{O}(n)$.

Since we insert at most $\sqrt{n}/11$ pairs in $\mathcal{Q}$ but there may be up to $3\sqrt{n}$ high-frequency pairs, once $\mathcal{Q}$ is empty we may need to fill it again with new high-frequency pairs (`while` loop at line 3 of Algorithm 2). We need to repeat this process at most $(3\sqrt{n})/(\sqrt{n}/11) \in \mathcal{O}(1)$ times, so the number of rounds is constant.

We call $\mathcal{Q}.min()$ in two cases: (i) after extracting the maximum from $\mathcal{Q}$ (Line 2, Algorithm 1), and (ii) within procedure $\mathcal{Q}.synchronize$, after discovering a new high-frequency pair XY and inserting it in $\mathcal{Q}$. Case (i) cannot happen more than $3\sqrt{n}$ times. As for case (ii), note that a high-frequency pair can be inserted at most once per round in $\mathcal{Q}$ within procedure $\mathcal{Q}.synchronize$. Since the overall number of rounds is constant and there are at most $3\sqrt{n}$ high-frequency pairs, also in this case we call $\mathcal{Q}.min()$ at most $\mathcal{O}(\sqrt{n})$ times. Overall, the time spent inside $\mathcal{Q}.min()$ is therefore $\mathcal{O}(n)$.

Finally, considerations of Section 3.3 imply that scanning/sorting occurrences lists inside operation $\mathcal{Q}.synchronize$ take overall linear time thanks to our amortization policy.

4.4 Low-Frequency Pairs Queue

We describe two low-frequency queue variants, denoted in what follows as *fast* and *light*. We start with the fast variant.

Fast queue

Let $0 < \epsilon \leq 1$ be a parameter chosen in advance. Our fast queue has maximum capacity $(\epsilon/13) \cdot n$ and is implemented with three components:

(i) **Set of doubly-linked lists** B. This is a set of lists; each list is associated to a distinct frequency. B is implemented as an array of elements of the form $\langle ab, P_{ab}, L_{ab}, F_{ab}, Prev_{ab}, Next_{ab} \rangle$, where:
- ab, P_{ab}, L_{ab}, and F_{ab} have the same meaning as in the high-frequency queue
- $Prev_{ab}$ points to the previous B element with frequency F_{ab} (NULL if this is the first such element)
- $Next_{ab}$ points to the next B element with frequency F_{ab} (NULL if this is the last such element)

Every B element takes 7 words. We allocate $\epsilon \cdot (7/13) \cdot n$ words for B (maximum capacity: $(\epsilon/13) \cdot n$)
(ii) **Doubly-linked frequency vector** $\mathcal{F}$. This is a word vector $\mathcal{F}[0, \dots, \sqrt{n}/3 - 1]$ indexing all possible frequencies of low-frequency pairs. We say that $\mathcal{F}[i]$ is *empty* if i is not the frequency of any pair in T. In this case, $\mathcal{F}[i] = NULL$. Non-empty $\mathcal{F}$'s entries are doubly-linked: we associate to each $\mathcal{F}[i]$ two values $\mathcal{F}[i].prev$ and $\mathcal{F}[i].next$ representing the two non-empty pair's frequencies immediately smaller/larger than i. We moreover keep two variables MAX and MIN storing the largest and smallest frequencies in $\mathcal{F}$. If i is the frequency of some character pair, then $\mathcal{F}[i]$ points to the first B element in the chain associated with frequency i: $B[\mathcal{F}[i]] = \langle ab, P_{ab}, L_{ab}, i, NULL, Next_{ab}\rangle$, for some pair ab. $\mathcal{F}$ takes overall $\sqrt{n}$ words of space
(iii) **Hash** $\mathcal{H}$. We keep a hash table $\mathcal{H} : \Sigma^2 \rightarrow [0, n]$ with $\epsilon \cdot (2/13) \cdot n$ entries. The hash is indexed by character pairs. $\mathcal{H}$ will be filled with at most $\epsilon \cdot n/13$ pairs (hash load ≤ 0.5). Collisions are solved by linear probing. The overall size of the hash is $\epsilon \cdot (6/13) \cdot n$ words: 3 words (one pair and one integer) per hash entry. $\mathcal{H}$'s entries point to B's entries: if ab is in the hash, then $B[\mathcal{H}[ab]] = \langle ab, P_{ab}, L_{ab}, F_{ab}, Prev_{ab}, Next_{ab}\rangle$

Overall, $\mathcal{Q} = \langle B, \mathcal{F}, \mathcal{H}\rangle$ takes $\sqrt{n} + \epsilon \cdot n$ words of space. Figure 2 depicts our low-frequency queue.

First, we show how function $new\_low\_freq\_queue(T, TP)$ is implemented. We build list $\langle k_1, j_1\rangle, \dots, \langle k_t, j_t\rangle$ sorted by frequency as done for the high-frequency queue. We scan this list left-to-right and insert at most $n/13$ low-frequency pairs in $\mathcal{H}$, starting from the most frequent ones. At the same time, we fill $\mathcal{F}$'s list pointers: while processing $\langle k_i, j_i\rangle$, if the maximum queue capacity has not been reached then we set $\mathcal{F}[k_{i-1}].next \leftarrow k_i$, and $\mathcal{F}[k_i].prev \leftarrow k_{i-1}$. We re-build TP (re-using the memory already allocated for it), and scan it left-to-right. For each maximal $TP[p, \dots, p+k-1]$ corresponding to a pair ab, if ab is in $\mathcal{H}$ then, in order:

1. we set $\mathcal{H}[ab] \leftarrow |B|$
2. if $\mathcal{F}[k] \neq NULL$ we set the fifth component of $B[\mathcal{F}[k]]$ ($Prev$ field) to $|B|$
3. we append $\langle ab, p, k, k, NULL, \mathcal{F}[k]\rangle$ at the end of B
4. we set $\mathcal{F}[k] \leftarrow |B| - 1$ (decreased by one because we just increased B's size).

This procedure runs in $\mathcal{O}(n)$ time and uses $\mathcal{O}(1)$ words of space in addition to T, TP, B, $\mathcal{H}$, and $\mathcal{F}$. Operations on $\mathcal{Q}$ are implemented as follows:

- $\mathcal{Q}[ab]$, $ab \in \bar{\Sigma}^2$: return second, third, and fourth components of $B[\mathcal{H}[ab]] = \langle ab, P_{ab}, L_{ab}, F_{ab}, Prev_{ab}, Next_{ab}\rangle$. $\mathcal{O}(1)$ expected time.
- $\mathcal{Q}.max()$: return the first component of $B[\mathcal{F}[MAX]]$. $\mathcal{O}(1)$ time.
- $\mathcal{Q}.min()$: return the first component of $B[\mathcal{F}[MIN]]$. $\mathcal{O}(1)$ time.

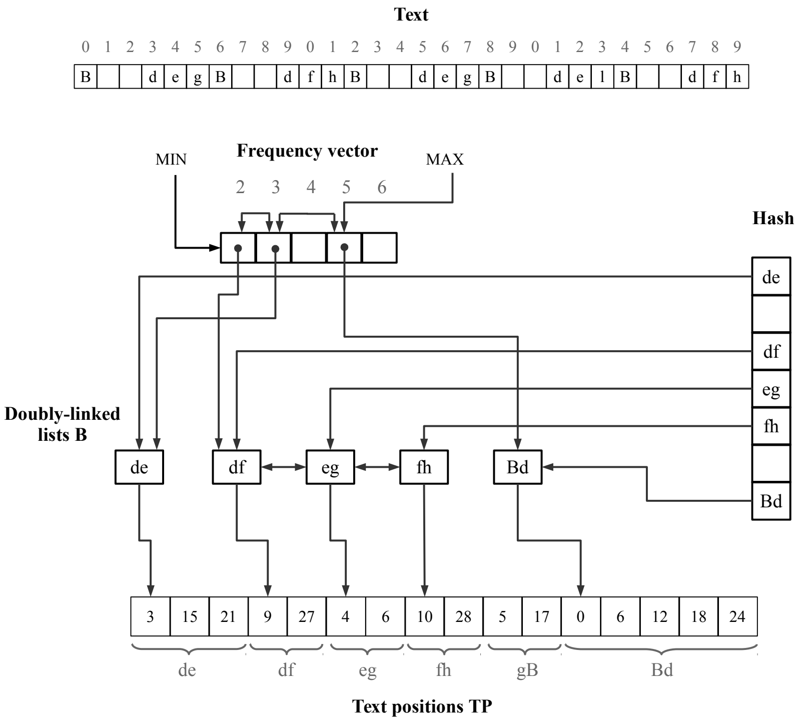

Figure 2: The low-frequency queue is composed by three components: frequency vector, hash, and doubly-linked lists. In this example, the text is obtained from `abcdegabcdfhabcdegabcdelabcdfh` after the substitutions (in order) $A \rightarrow ab$ and $B \rightarrow Ac$. Pairs de, df, eg, fh, and Bd have been inserted in the queue. We do not show F- and L-fields in B's elements to improve readability.

- $\mathcal{Q}.remove(ab)$: delete $B[\mathcal{H}[ab]]$ from its linked list in B and delete ab from $\mathcal{H}$. If ab's linked list in B is now empty (i.e. ab was the last pair with frequency F_{ab}), re-compute new MAX and MIN if necessary (by using $\mathcal{F}$'s list pointers), remove frequency F_{ab} from the linked list in $\mathcal{F}$ and set $\mathcal{F}[F_{ab}] \leftarrow NULL$. $\mathcal{O}(1)$ expected time.

- $\mathcal{Q}.contains(ab)$. This operation requires one access to $\mathcal{H}$. $\mathcal{O}(1)$ expected time.

- $\mathcal{Q}.size()$: we only need to keep a variable storing the current size and update it at each remove operation. $\mathcal{O}(1)$ time.

- $\mathcal{Q}.decrease(AB)$: decrease the fourth component of $B[\mathcal{H}[AB]]$ (i.e. F_{AB}) by one. Remove $B[\mathcal{H}[AB]]$ from its linked list in B and insert it in the list having $\mathcal{F}[F_{AB}]$ as first element (if $\mathcal{F}[F_{AB}] = NULL$, then create a new linked list and adjust $\mathcal{F}$'s list pointers). Re-compute MAX and MIN if necessary (by using

$\mathcal{F}$'s list pointers). If F_{AB} becomes equal to 1, remove the pair from the queue. $\mathcal{O}(1)$ expected time.

- $\mathcal{Q}.synchronize(AB)$. Let $\overline{ab}$ be a global variable storing the pair with the smallest frequency in $\mathcal{Q}$ (Algorithm 1, Line 2). We apply the sorting algorithm of Section 4.2 to sub-array $TP' = TP[P_{AB}, \dots, P_{AB} + L_{AB} - 1]$, excluding text positions that contain a blank character. After this, positions are clustered in TP' by character pairs. For each contiguous maximal sub-array $TP[p, \dots, p + k - 1]$ of TP' corresponding to a pair XY:

  - If $XY \neq AB$ and $k > F_{\overline{ab}}$, we remove $\overline{ab}$ and insert XY in our priority queue as follows. Let $j_{min} = \mathcal{H}[\overline{ab}]$. We call $\mathcal{Q}.remove(\overline{ab})$, overwrite $B[j_{min}] \leftarrow \langle XY, p, k, k, NULL, NULL \rangle$, insert $B[j_{min}]$ in the linked list having $\mathcal{F}[k]$ as first element (assigning a value to $Next_{XY}$) or create a new list if $\mathcal{F}[k] = NULL$, and insert $\mathcal{H}[XY] \leftarrow j_{min}$ in the hash. We re-compute MIN taking the least frequent pair between XY and the pair corresponding to $B[\mathcal{F}[MIN]]$, and re-compute the minimum $\overline{ab}$ (i.e. the pair corresponding to $B[\mathcal{F}[MIN]]$).
  - If $XY = AB$, then we just overwrite $B[\mathcal{H}[AB]] \leftarrow \langle AB, p, k, k, Prev_{AB}, Next_{AB} \rangle$.

  Running time: $\mathcal{O}(L_{AB})$, where L_{AB} is AB's interval length at the moment of entering in this procedure.

Time complexity

Since we insert at most $(\epsilon/13) \cdot n$ pairs in $\mathcal{Q}$ but there may be up to $\mathcal{O}(n)$ low-frequency pairs, once $\mathcal{Q}$ is empty we may need to fill it again with new low-frequency pairs (`while` loop at line 10 of Algorithm 2). We need to repeat this process $\mathcal{O}(n/(n \cdot \epsilon/13)) \in \mathcal{O}(1/\epsilon)$ times before all low-frequency pairs have been processed. Since operations at Lines 11, 12, 15, and 16 take $\mathcal{O}(n)$ time, the overall time spent inside these procedures is $\mathcal{O}(n/\epsilon)$. Using the same reasonings of the previous section, it is easy to show that the time spent inside $\mathcal{Q}.synchronize$ is bounded by $\mathcal{O}(n)$ thanks to our amortization policy. Moreover, since all queue operations except $\mathcal{Q}.synchronize$ take constant time, we spend overall $\mathcal{O}(n)$ time operating on the queue. These considerations imply that instructions in Lines 10-16 of Algorithm 2 take overall $\mathcal{O}(n/\epsilon)$ randomized time. Theorem 1(i) follows.

Light queue

While for the fast queue we reserve $\epsilon \cdot n$ space for B and $\mathcal{H}$, in the light queue we observe that we can re-use the space of *blank text characters* generated after replacements. The idea is the following. Let S_i be the capacity (in terms of number of pairs) of the queue at the i-th execution of the `while` loop at Line 10; at the beginning, $S_1 = 1$. After executing operations at Lines 11-16, new blanks are generated and this space is available at the end of the memory allocated for the text, so we can accumulate it on top of S_i obtaining space $S_{i+1} \geq S_i$. At the next execution of the `while` loop, we fill the queue until all the available space S_{i+1} is filled. We proceed like this until all pairs have been processed. The question is: how many times the `while` loop at Line 10 is executed?

proof1 I found this alternative proof, which is much simpler (is it correct?) Replacing a pair ab generates at least $F_{ab}/2$ blanks: in the worst case, the pair is of the form aa and all pair occurrences overlap, e.g. in $aaaaaa$ (which generates 3 blanks). Note that $F_{ab} \geq 2$ (otherwise we do not consider ab for substitution). This implies that replacing a pair ab generates at least $F_{ab}/2 \geq 2/2 = 1$ blanks: after replacing all S_i pairs at round i, we have at least S_i blanks in the text. Recall that a pair takes 13 words to be stored in our queue. At round $i + 1$ we therefore have room for a total of $S_{i+1} \geq (1 + 1/13)S_i$ new pairs. This implies $S_i \geq (1+1/13)^{i-1}$. Since $S_i \leq n$ for any i, the number R of rounds can be computed as $S_R \leq n \Leftrightarrow (1+1/13)^{R-1} \leq n \Leftrightarrow R \in \mathcal{O}(\log n)$.

proof2 Replacing a pair ab generates at least $F_{ab}/2$ blanks: in the worst case, the pair is of the form aa and all pair occurrences overlap, e.g. in $aaaaaa$ (which generates 3 blanks). Moreover, replacing a pair with frequency F_{ab} decreases the frequency of at most $2F_{ab}$ pairs in the active priority queue (these pairs can therefore disappear from the queue). Note that $F_{ab} \geq 2$ (otherwise we do not consider ab for substitution).

After one pair ab is replaced at round i, the number M_i of elements in the active priority queue is at least $M_i \geq S_i - (1 + 2F_{ab})$. Letting $f_1, f_2, \ldots$ be the frequencies of all pairs in the queue, we get that after replacing all elements the number (0) of elements in the priority queue is:

$$0 \geq S_i - (1 + 2f_1) - (1 + 2f_2) - \cdots$$

which yields

$$S_i \leq (1 + 2f_1) + (1 + 2f_2) + \cdots$$

since $f_i/2 \geq 1$ then

$$S_i \leq 2.5f_1 + 2.5f_2 + \cdots = 2.5\sum_i f_i$$

so when the active priority queue is empty we have at least $\sum_i f_i/2 \geq S_i/5$ new blanks. Recall that a pair takes 13 words to be stored in our queue. In the next round we therefore have room for a total of $(1 + 1/(5 \cdot 13))S_i = (1 + 1/65)S_i$ new pairs. This implies $S_i = (1 + 1/65)^{i-1}$. Since $S_i \leq n$ for any i, the number R of rounds can be computed as $S_R \leq n \Leftrightarrow (1 + 1/65)^{R-1} \leq n \Leftrightarrow R \in \mathcal{O}(\log n)$.

With the same reasonings used before to analyze the overall time complexity of our algorithm, we get Theorem 1(ii).

5 References

[1] N Jesper Larsson and Alistair Moffat, "Off-line dictionary-based compression," *Proceedings of the IEEE*, vol. 88, no. 11, pp. 1722–1732, 2000.

[2] Raymond Wan, *Browsing and Searching Compressed Documents*, Ph.D. thesis, Department of Computer Science and Software Engineering, University of Melbourne., 1999.

[3] Rodrigo González and Gonzalo Navarro, "Compressed text indexes with fast locate," in *Proc. 18th CPM*, 2007, pp. 216–227.

[4] Francisco Claude and Gonzalo Navarro, "Fast and compact web graph representations," *ACM Trans. Web*, vol. 4, no. 4, pp. 16:1–16:31, 2010.

[5] Gonzalo Navarro and Luís Russo, "Re-pair achieves high-order entropy," in *Proc. 18th DCC*, 2008, p. 537.

[6] Gianni Franceschini, S. Muthukrishnan, and Mihai Patrascu, "Radix sorting with no extra space," in *Proc. 15th ESA*, 2007, pp. 194–205.

[7] Jeffrey S. Salowe and William L. Steiger, "Simplified stable merging tasks," *J. Algorithms*, vol. 8, no. 4, pp. 557–571, 1987.